Statistics on Query Expressions
in Relational Database Management Systems

Nicolas Bruno

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY
2003
ABSTRACT

Statistics on Query Expressions
in Relational Database Management Systems

Nicolas Bruno

The query optimizer is the component in a relational database system that identifies efficient execution plans for input queries. Modern optimizers generally explore many alternative query plans in a cost-based manner. Specifically, the resource consumption and associated cost of each candidate plan is estimated, and the plan with the least expected cost is chosen for execution. The cost estimation for a plan depends on several factors, including resource availability during execution, the specific operators that compose the plan, and the size of intermediate results that would be generated during the plan execution. Among these factors, the intermediate-result size (or cardinality) estimation is the main source of inaccuracies during optimization: cardinality estimation typically relies on several simplifying assumptions that often do not hold in practice. Optimizers then sometimes base their decisions on inaccurate information and produce low-quality execution plans. To address this limitation, in this thesis we introduce the concept of SITs, which are statistics built on query expressions. SITs directly and accurately model intermediate results in a query execution plan, and therefore avoid error-prone simplifying assumptions during cardinality estimation. If optimizers have appropriate SITs available during optimization, the resulting query plans can be dramatically better than otherwise. Although SITs are a fairly simple concept, challenging problems need to be addressed before SITs can be seamlessly integrated into modern relational database systems. In this thesis we study three important challenges associated with SITs. First, we show how to modify query optimizers to exploit the additional statistical information provided by SITs without significantly increasing optimization time. Second, we study a spectrum
of alternatives to create SITs, which balance efficiency of construction and accuracy of
the resulting estimators. Third, we present techniques to recommend a small but highly
beneficial set of SITs to materialize in a database system for a given query workload. In
summary, we address the main obstacles for enabling SITs for optimization, namely which
SITs to build, how to build them, and how to exploit them during optimization. Overall,
SITs constitute a well-founded approach for dealing with complex data correlations, and
positively impact the efficiency of relational database systems.
Contents

1 Introduction 1

2 Optimization in Relational Database Systems 8
  2.1 Architecture of a Query Optimizer ............................ 10
  2.2 Cardinality Estimation for Optimization .......................... 13
    2.2.1 Histograms ............................................. 14
      2.2.1.1 Unidimensional Histograms .......................... 14
      2.2.1.2 Multidimensional Histograms .......................... 17
    2.2.2 Cardinality Estimation using Histograms ...................... 20
      2.2.2.1 Selection Queries .................................... 21
      2.2.2.2 Join Queries ....................................... 23
      2.2.2.3 Select-Project-Join Queries ......................... 25
      2.2.2.4 Queries with Group-by Clauses ....................... 26
    2.2.3 Cardinality Estimation in the Absence of Histograms ........ 27
  2.3 Conclusion ............................................... 28

3 SITs: Statistics on Query Expressions 29
  3.1 Simplifying Assumptions in Cardinality Estimation ............... 30
    3.1.1 Independence Assumption .................................. 31
    3.1.2 Containment Assumption .................................. 33
    3.1.3 Sampling Assumption ..................................... 33
### 3.1.4 Other Assumptions

3.2 SITs: Avoiding Error Propagation

3.2.1 A Validating Experiment

3.2.2 Exploiting SITs: Challenges

3.3 Conclusion

4 Exploiting SITs

4.1 A Wrapper-based Architecture for Exploiting SITs

4.1.1 Transformation Step

4.1.1.1 Input Plan Analysis

4.1.1.2 Application of Relevant SITs

4.1.1.3 Incomplete SIT-Sets and Auxiliary SITs

4.1.1.4 Putting it All Together

4.1.2 Actual Estimation

4.1.3 Conclusion

4.2 A Formal Framework for Exploiting SPJ SITs

4.2.1 Conditional Selectivity

4.2.1.1 Atomic Decompositions

4.2.1.2 Separable Decompositions

4.2.2 Obtaining the Most Accurate Selectivity Estimation

4.2.2.1 Ranking Candidate Decompositions

4.2.2.2 Algorithm \texttt{getSelectivity}

4.2.2.3 Candidate SITs

4.2.2.4 Error Functions

4.2.2.5 A Final Example

4.2.3 Integration with an Optimizer

4.2.3.1 Cascades-based Optimization

4.2.3.2 Integrating \texttt{getSelectivity} with Cascades

4.2.4 Extensions

---

43
4.2.4.1 Non-Conjunctive SPJ Queries 79
4.2.4.2 SPJ Queries with Group-by Clauses 79
4.2.4.3 Other Extensions 83

4.3 Experimental Evaluation 85
4.3.1 Comparison between the Wrapper Approach and $getSelectivity$ 87
4.3.2 Accuracy of $getSelectivity$ 88
4.3.3 Efficiency of $getSelectivity$ 89
4.3.4 Exploiting Multidimensional Statistics using $getSelectivity$ 92

4.4 Conclusion 94

5 Building SITs
5.1 A Feedback-Driven Approach to Build and Refine SITs 97
5.1.1 STHoles Histograms 100
5.1.2 STHoles Construction Algorithm 102
5.1.2.1 Identifying Candidate Holes 103
5.1.2.2 Drilling Candidate Holes as New Histogram Buckets 105
5.1.2.3 Merging Buckets 107
5.1.3 Experimental Evaluation 111
5.1.3.1 Comparison of STHoles and other Histogram Techniques 115
5.1.3.2 Experiments Specific to Histogram Refinement 118

5.2 An Off-line Approach to Build SITs 121
5.2.1 Review of the Main Assumptions behind Cardinality Estimation 123
5.2.2 Creating a Single SIT 124
5.2.2.1 Architecture of $Sweep$ 124
5.2.2.2 Balancing Accuracy and Efficiency 127
5.2.2.3 Beyond Two-way Join Generating Queries 128
5.2.2.4 Other Extensions 130
5.2.2.5 Summary 132
5.2.3 Creating Multiple SITs 132
List of Figures

1.1 The RDBMS chooses efficient execution plans to evaluate input queries. . . 2

2.1 Executing SQL queries in a relational database system. . . . . . . . . . . . 9
2.2 Simplified architecture of the query optimizer in a database system. . . . 10
2.3 Query execution plans for various instances of a template query. . . . . . 12
2.4 Histograms approximate the value distribution of columns in a database. . 16
2.5 Multidimensional histograms approximate joint distributions of values. . . 19
2.6 Uniform spread and frequency assumption for histogram buckets. . . . . . 20
2.7 Cardinality estimation for range queries using histograms. . . . . . . . . . 21
2.8 Cardinality estimation of join predicates using histograms. . . . . . . . . . 24
2.9 Propagation of histograms in SPJ queries. . . . . . . . . . . . . . . . . . . . 26

3.1 Assumptions to estimate the cardinality of an SPJ query. . . . . . . . . . 30
3.2 Effect of the independence assumptions during cardinality estimation. . . 32
3.3 Using statistics over views to improve cardinality estimation. . . . . . . . 35
3.4 Query execution plans for different sets of available SITs. . . . . . . . . . 38
3.5 By using SITs, execution times can be drastically reduced. . . . . . . . . 39

4.1 A wrapper intercepts cardinality requests to exploit SITs. . . . . . . . . . 44
4.2 Transformation step in the modified cardinality estimation module. . . . . 46
4.3 Transforming an input query plan to exploit SITs. . . . . . . . . . . . . . 50
4.4 Exploiting SITs using materialized view matching technology. . . . . . . 51
5.17 Normalized absolute error after updates. ............................................. 120
5.18 Useful SITs cannot always be extracted from execution plans. ................. 122
5.19 Creating SIT($S.a|R \bowtie_{R.x=S.y} S$) using Sweep. ......................... 125
5.20 Using Sweep to approximate a linear-join query. ................................. 129
5.21 SIT with acyclic-join query and join tree for U.a. ................................. 129
5.22 The graph induced by $R = \{abcd, bca\}$. ........................................... 135
5.23 Solving the shortest common supersequence problem using A*. ............... 137
5.24 Dependency sequences for join generating queries. ............................... 138
5.25 Obtaining successor nodes in the modified A* algorithm. ...................... 141
5.26 Creating SITs with skewed join generating queries. ............................. 144
5.27 Creating SITs with varying numSits. ................................................ 146
5.28 Creating SITs with varying lenSits. .................................................. 147
5.29 Creating SITs with varying number of tables. .................................... 148
5.30 Creating SITs with varying memory limit. ......................................... 148

6.1 Workload information used to select SITs in an RDBMS. ......................... 155
6.2 Architecture of the SIT selection tool. ................................................. 157
6.3 Generating candidate SITs for a query. .............................................. 159
6.4 Generating candidate SITs for a workload. ........................................ 160
6.5 Alternatives to propagate a histogram through a predicate. .................... 164
6.6 Cardinality and selectivity estimation using different strategies. .............. 165
6.7 The area between $F_{MIN}$ and $F_{MAX}$ indicates the possible change in selectivity introduced by a given predicate. ....................... 168
6.8 Estimating Diff using Area values. .................................................. 169
6.9 Recommending SITs for a workload. ................................................ 172
6.10 Evaluation of the top-$k$ recommended SITs for varying $k$. ................. 174
6.11 Execution time of validation queries optimized with no SITs vs. with the top-30 SITs ................................................................. 176
6.12 Evaluation of different components of the what-if module. .................. 179
8.1 Integrating the different components introduced in the thesis. . . . . . . 191
Acknowledgments

This dissertation would not have been possible without the continuous support and guidance of Luis Gravano and Surajit Chaudhuri. Luis, my advisor at Columbia University, gave me absolute freedom to choose research topics, offered me advice whenever I needed it the most, and taught me how to think and communicate clearly. His extremely detailed comments on this thesis contributed to make it crisper and more organized. Surajit, my mentor at Microsoft Research, always encouraged me to focus on the right problems, and offered critical feedback that helped improve early versions of this thesis. I am grateful for many technical and non-technical discussions I had with both Luis and Surajit over these years. I would also like to thank Nick Koudas and Divesh Srivastava, from AT&T Research, for giving me the opportunity to work with them on exciting research projects. In addition to Luis, Surajit, and Divesh, Ken Ross and Alfred Aho were part of my dissertation committee. They all gave me quality feedback that allowed me to clarify and improve various aspects of this dissertation. The DMX group at Microsoft Research and the Database group at Columbia University are terrific research environments, and I am grateful to all the members of both groups for many valuable discussions and interactions. I had the pleasure to meet many wonderful friends both in New York and Seattle, who helped me achieve the right balance between work and fun. Finally, I thank my family for encouraging me to come to grad school, and my wife Pía for her non-stop support, patience and tremendous help during all these years.
To Pía and Tomás
Chapter 1

Introduction

Relational database management systems (RDBMSs) let users specify queries using high-level declarative languages such as SQL. Users define their desired results without detailing how such results must be obtained. The query optimizer, a component of the RDBMS, is then responsible for finding an efficient procedure, or execution plan, to evaluate the input query. For that purpose, the optimizer searches a large space of alternative execution plans, and chooses the one that is expected to be evaluated in the least amount of time (see Figure 1.1). Once the optimizer produces the execution plan estimated to be the best plan, the execution engine of the RDBMS provides an environment in which this plan is evaluated. State-of-the-art query optimizers search the space of alternative execution plans in a cost-based manner. Conceptually, modern optimizers assign each candidate plan its estimated cost (i.e., the expected amount of resources that the execution engine would require to evaluate the candidate plan), and choose the plan with the least expected cost for execution.

Example 1 Consider tables Population and States in a US census database. Table Population has several attributes with information about each US resident, such as gender, level of studies, and city and state of residence. In turn, table States contains one row per US state, and each row has several attributes such as area in square miles, gross state product, and unemployment rate. The following SQL query extracts information
about all residents in states that have a gross product exceeding 500 billion dollars:

```
SELECT P.age, P.gender, ...
FROM Population P, States S
WHERE P.state_id = S.state_id AND
     S.gross_product > 500,000,000,000
```

Note that the query above only indicates the structure and relationships that the output tuples must satisfy, without specifying how to obtain the required information. In general, there are many possible execution plans to obtain the same result. For instance, we can first join tables `Population` and `States` on `state_id` and then filter out the joined tuples that do not satisfy the condition `S.gross_product > 500 billion`. Alternatively, we can use an index on `States.gross_product` to quickly identify the states with gross product exceeding 500 billion and then join each resulting state with all the matching tuples in `Population`. Which alternative plan is the most efficient depends on the actual data distribution of tables `States` and `Population`. The role of the optimizer is then to
estimate the cost of each plan to evaluate the input query, and choose the candidate plan with the smallest estimated execution cost. ■

The estimated cost of a plan depends on several factors (see [GMUW00]). The most important among these factors is the size of intermediate results that would be generated during the execution of the plan. For instance, the running time of specific join algorithms (e.g., nested-loop joins or hash joins) crucially depends on the size of the join input relations, which in general are arbitrary intermediate results. Traditionally, query optimizers use statistics that are built over base tables to estimate intermediate-result sizes. In doing so, optimizers rely on a number of simplifying assumptions to propagate these base-table statistics through complex query plans (see Chapter 2 for a discussion on query optimization in relational databases and current mechanisms for exploiting statistics for cardinality estimation). Unfortunately – but not unexpectedly – such result-size estimates often are off by orders of magnitude.

Example 1 (cont.) Suppose we use the actual US population and state information for the year 2002 [Uni02a, Uni02b]. To estimate the result of the query above, the optimizer first uses information about table constraints to determine that each resident from Population matches one and only one state in States (i.e., the query combines tables Population and States by using a foreign-key join). The result of this join would be accurately estimated as the number of tuples in Population, or 281,421,906 in our example. To also take into account the effect of predicate $S.\text{gross\_product} > 500$ billion, the optimizer first estimates, using statistics on column States.\text{gross\_product}, that 3 out of the 50 states have a gross product exceeding 500 billion dollars. Finally, the optimizer assumes independence between the distribution of people across states and the gross product of each state, and extrapolates the reduction factor 3/50, which is correct for table States in isolation, to the whole query result. In summary, the number of US residents living in states with gross products exceeding 500 billion dollars is estimated as $281,421,906 \cdot 3/50 = 16,885,314$, or less than 17 million people. However, the correct
number in our example is 73,699,925, or over 73 million people, which is more than four times the estimated value. In fact, the three states that satisfy the filter condition on column States.gross_product are California, New York, and Texas. Not surprisingly, these are also the states with the largest populations, and therefore the calculated value is a severe underestimate of the correct result.

As illustrated by the example above, traditional mechanisms to manipulate base-table statistics may result in highly inaccurate cardinality estimates. This is a serious problem because the optimizer might choose low-quality execution plans relying on inaccurate information. In fact, the set of operators in the best execution plan to evaluate the query above (or other more complex queries that contain the query above) can be very different if some intermediate result size varies from 17 to 73 million tuples. To address this limitation, we could maintain additional statistical information in the RDBMS. In our example, we could store an approximation of the gross state product distribution weighted by the number of people in each state. Using this additional statistical information, we could then obtain an accurate estimation of the result of the query in Example 1 without making any independence assumption between attributes. The optimizer, exploiting more accurate information, would then obtain a better-quality execution plan for the query.

In Chapter 3, we generalize this idea by introducing the concept of SITs, which are statistics on query expressions. SITs can be used to accurately model the distribution of tuples on intermediate nodes in a query execution plan, therefore alleviating the error-prone propagation of base-table statistics. We then quantify the potential benefits of exploiting SITs during optimization with a simple experiment. Specifically, we show that when exact cardinality information for all intermediate query sub-plans is available during optimization, the resulting execution plans can be drastically improved, with execution times that are more than an order of magnitude faster that those obtained when only base-table statistics are available.

The obvious acronym SQE (Statistics on Query Expressions) is not quite as nice as SIT (Statistics on Intermediate Tables). We decided to pick a nicer acronym rather than being technically accurate.
Despite the conceptual simplicity of SITs, we need to address significant challenges before SITs can be effectively used in existing database systems. First, we must show how query optimizers can be adapted to exploit the additional statistical information provided by SITs. This is not an easy task, because multiple SITs can be useful to only portions of the input query, and some SITs can be mutually incompatible. In Chapter 4 we address this problem and introduce techniques to carefully use available SITs for cardinality estimation during query optimization. We first develop a general framework to exploit SITs that intercepts cardinality requests made by the optimizer and transforms the input query plan into an equivalent one that exploits SITs. The transformation step leverages work done in the context of materialized view matching, and therefore our techniques can be easily extended to incorporate new results from the literature. Then, we focus on the family of select-project-join queries with optional group-by clauses, and introduce a formal framework to reason with SITs. Using this framework, we can efficiently search the complete space of opportunities to estimate cardinality values by exploiting available SITs. We design a dynamic programming algorithm to efficiently return the most accurate cardinality estimate for an input query. The algorithm considers many critical alternatives that are otherwise missed by techniques that rely exclusively on materialized view matching technology, and can be easily integrated with existing relational optimizers. Our experiments show that the proposed techniques impose very little overhead to existing optimizers, and return cardinality estimates that are much more accurate than those obtained by state-of-the-art optimizers that do not consider SITs for cardinality estimation.

A second challenge is to provide mechanisms to efficiently build SITs in a database system. While simple approaches might be sufficient in some cases, other situations require the use of customized algorithms. In particular, if efficiency is critical, we need to develop techniques that can build approximate versions of SITs in a fraction of the time it takes to fully evaluate their generating query expressions, but without significantly sacrificing accuracy. In Chapter 5 we explore alternative algorithms to create SITs. The
new techniques are more efficient than naive approaches that materialize intermediate query expressions, at the expense of some precision. We first propose a feedback-based architecture to build and refine SITs on the fly as queries are answered in an RDBMS. Specifically, we intercept intermediate results that are passed between operators as query plans are executed, and gather aggregated information that is later used to refine SITs. This approach does not access the underlying data distribution directly to build SITs, but instead SITs are incrementally refined in parallel with actual query executions, imposing low overhead to the query processor. We then study a complementary family of techniques to build SITs off-line whenever the feedback-driven procedure is not adequate. Specifically, we introduce techniques to create SITs that effectively work on close approximations of the SITs’ query expressions. We show experimentally that our techniques can be used to build SITs efficiently (using a single scan over the referenced tables) without significantly compromising accuracy. Finally, we recognize that while creating several SITs at once, commonalities between their query expressions can be leveraged. We study the problem of creating multiple SITs simultaneously by carefully sharing portions of the computation to create each individual SIT (e.g., sequential scans over base tables). We map this problem to a generalization of the “Shortest Common Supersequence” problem, and propose both exact and approximate algorithms to obtain good schedules to create multiple SITs. We then show that our approach obtains close-to-optimal schedules to create multiple SITs simultaneously.

A third challenge consists of automatically identifying useful SITs to materialize in an RDBMS, which would free database administrators from the burden of choosing which SITs to maintain. In general, a complex SIT can greatly benefit one particular (complex) query, but it may be useless for even slightly different queries. In contrast, a simpler SIT might be moderately useful for a larger class of queries. It is then important to provide techniques that recommend SITs with the right balance between specificity and generality. In Chapter 6 we present an algorithm to recommend a relatively small but useful set of single-column SITs with select-project-join (SPJ) generating queries to materialize in an
RDBMS. Similar to previous work on automatic index and materialized view selection for database systems, our algorithms use workload information and query feedback to recommend SITs. For efficiency, the SIT recommendation algorithm proceeds without materializing any candidate SIT in advance, which introduces additional challenges. To still be able to analyze the potential impact of candidate SITs, we rely on a what-if analysis module. This module estimates the expected error in cardinality estimation if candidate SITs under consideration were present during estimation. For that purpose, we propose techniques to quantify the expected departure from independence of the SITs’ generating query results. The proposed technique works closely with the optimizer, hence assuring that the recommended SITs will be used by the optimizer during subsequent cardinality estimation requests. We conclude the chapter by reporting an experimental evaluation of our proposed algorithm to recommend SITs (and also the framework of Chapter 4 to exploit available SITs) over a commercial RDBMS.

Chapters 7 and 8 conclude this dissertation by discussing related work and reviewing the main contributions of this thesis.
Chapter 2

Query Optimization in Relational Database Systems

Relational query languages provide a high-level declarative interface to access data stored in relational database systems. With a declarative language, users (or applications acting as users) write queries stating what they want, but without specifying step-by-step instructions on how to obtain such results. In turn, the RDBMS internally determines the best way to evaluate the input query and obtains the desired result. Structured Query Language, or SQL [MS93], has become the most widely used relational database language. To answer a given SQL query, a typical RDBMS goes through a series of steps, illustrated in Figure 2.1:

1. The input query, treated as a string of characters, is parsed and transformed into an algebraic tree that represents the structure of the query. This step performs both syntactic and semantic checks over the input query, rejecting all invalid requests.

2. The algebraic tree is optimized and turned into a query execution plan. A query execution plan indicates not only the operations required to evaluate the input query, but also the order in which they are performed, the algorithm used to perform each step, and the way in which stored data is obtained and processed [Gra93].
3. The query execution plan is evaluated and results are passed back to the user in the form of a relational table.

Modern relational query optimizers are complex pieces of code and typically represent 40 to 50 developer-years of effort [RG00]. As stated before, the role of the optimizer in a database system is to identify an efficient execution plan to evaluate the input query. To that end, optimizers usually examine a large number of possible query plans and choose the one that is expected to result in the fastest execution. In this chapter we describe the components of a generic query optimizer and show how statistical information can be used to improve the quality of plans returned by query optimizers.
2.1 Architecture of a Query Optimizer

Several query optimization frameworks have been proposed in the literature [GD87, GM93, SAC+79, HFLP89, Gra95] and most modern optimizers rely on the concepts introduced in these references. Although implementation details vary among specific systems, virtually all optimizers share the same basic structure [Ioa97, Cha98], shown in Figure 2.2.

For each input query, the optimizer considers a multiplicity of alternative plans. For that purpose, an enumeration engine navigates through the space of candidate execution plans by applying rules. Some optimizers have a fixed set of rules to enumerate alternative plans (e.g., [SAC+79]), while others implement extensible transformational rules to navigate through the search space (e.g., [HFLP89, Gra95]). During optimization, a cost module estimates the expected consumption of resources of each discovered query plan (resources are usually the number of I/O’s, but can also include CPU time, memory, communication bandwidth, or a combination of these). Finally, once all interesting execution plans are explored, the optimizer extracts the best one, which is evaluated in the execution engine (see Figure 2.1).
The cost estimation module is then a critical component of a relational optimizer. In general, it is not possible to obtain the exact cost of a given plan without executing it (which does not make sense during optimization). Thus, the optimizer is forced to estimate the cost of any given plan without executing it. It is then fundamental for an optimizer to rely on accurate procedures to estimate costs, since optimization is only as good as its costs estimates. Cost estimation must also be efficient, since it is repeatedly invoked during the optimization process. The basic framework for estimating costs is based on the following recursive approach described in [Cha98]:

1. Collect statistical summaries of stored data.

2. Given an operator in the execution plan and statistical summaries for each of its sub-plans, determine:
   
   (a) Statistical summaries of the output.
   
   (b) Estimated cost of executing the operator.

The second step can be applied iteratively to an arbitrary tree to derive the costs of each operator. The estimated cost of a plan is then obtained by combining the costs of each of its operators. In general, the number of disk I/O’s needed to manage intermediate results while executing a query plan (and thus the plan’s cost) is a function of the sizes of the intermediate query results [GMUW00, Section 7.4]. Therefore, the cost estimation module heavily depends on cardinality estimates of sub-plans generated during optimization. The following example illustrates how sizes of intermediate results can significantly change the plan that is chosen by an optimizer.

Example 2 Consider the following query template, where $C$ is a numeric parameter:

```
SELECT * FROM R,S
WHERE R.x=S.y and R.a<C
```

Figure 2.3 shows the execution plans produced by an optimizer when we instantiate $C$ with the values 20, 200, and 2000. Although the three instantiated queries are almost identical,
the resulting query plans are considerably different. For instance, in Figure 2.3(a), the optimizer estimates that the number of tuples in \( R \) satisfying \( R.a < 20 \) is very small, so it chooses to evaluate the query as follows. First, using a secondary index over \( R.a \), it retrieves the record identifiers of all tuples in \( R \) that satisfy \( R.a < 20 \). Then, using lookups against table \( R \), it fetches the actual tuples that correspond to those record identifiers. Finally, it performs a nested-loop join between the subset of tuples of \( R \) calculated before, and table \( S \), which is sequentially scanned. For the case \( C=2000 \) in Figure 2.3(c), the optimizer estimates that the number of tuples of \( R \) satisfying \( R.a < 2000 \) is rather large, and therefore chooses to scan both tables sequentially (discarding on the fly the tuples from \( R \) that do not satisfy the condition \( R.a < 2000 \)) and then perform a hash join to obtain the result. (In this scenario, the lookups of the previous plan would have been too numerous, and therefore, too expensive.) Finally, Figure 2.3(b) shows yet another execution plan that is chosen when the number of tuples of \( R \) satisfying the predicate is neither too small nor too large. In this case, table \( S \) is scanned in increasing order of \( S.y \) using a clustered index, and table \( R \) is first scanned sequentially (discarding invalid tuples on the fly as before) and then sorted by \( R.x \). Finally, a merge join is performed on the two intermediate results.
It is known that if cardinality estimates are accurate, overall cost estimates are typically off by no more than 10 percent [SLMK01]. However, cardinality estimates can be off by orders of magnitude when the underlying assumptions on the data distribution are invalid. Clearly, if the optimizer does not have accurate cardinality estimations during optimization, the “wrong” execution plan might be chosen for a given query. In the previous example, if the number of tuples satisfying $R.a < 2000$ is underestimated, the optimizer could choose the less efficient plan (for that scenario) of Figure 2.3(b), and therefore waste time by sorting a large intermediate subset of $R$. In the context of adaptive query processing [HFC+00], where initial bad choices during optimization can be later corrected during query execution, accurate cardinality estimates allow the optimizer to start with a higher quality execution plan, thus minimizing the probability of dynamic changes during query execution. For that reason, it is crucial to provide the optimizer with accurate procedures to estimate cardinality values during optimization. In the next section we give an overview of statistical structures that can be used to estimate the cardinality of intermediate results generated by query sub-plans during optimization.

2.2 Cardinality Estimation for Optimization

As explained in the previous section, the optimizer uses statistical information during optimization to obtain accurate cardinality estimates of intermediate tables produced by query sub-plans. For every table, statistical information usually includes the number of tuples, the average size of each tuple, and other values such as the number of physical pages used by the table, or sometimes the number of distinct tuples in the table. Statistical information on table columns, if available, can be helpful for estimating the cardinality of range or join predicates.

A large body of work in the literature studies the representation of statistics on a given column or combination of columns. The proposed techniques include wavelets [MVW98, CGRS00], discrete cosine transformations [LKC99], sampling [OR90, GGMS96, CMN99], sketches [AGMS99, TGIK02], fractal dimension concepts [FK97, BF95], parametric curve
fitting [CR94, SLRD93], and histograms [PSC84, MD88, PIHS96, PI97, JKM+98, AC99, 
KW99, GKTD00, BCG01a]. In this thesis we focus on histograms, which are the most 
common statistical summaries used in commercial RDBMS.

2.2.1 Histograms

Histograms divide the values of a column (or set of columns) into a number of buckets, 
and associate with each bucket some aggregated information. The number of buckets 
influences the accuracy of the histogram, but also affects memory usage, since relevant 
histograms are loaded into memory during optimization.

Definition 1  A histogram on attribute (or set of attributes) A, denoted $H(A)$, consists 
of a set of buckets $\{b_1, \ldots, b_k\}$. Each bucket $b_i$ represents a sub-range $r_i$ of A’s domain, 
and has at least two associated values. The frequency $f_i$ of bucket $b_i$ corresponds to the 
number of tuples $t$ in the data set for which $t.A \in r_i$, and the value $dv_i$ of bucket $b_i$ 
represents the number of distinct values of $t.A$ among all the tuples $t$ for which $t.A \in r_i$.

Both the particular procedure used to select bucket boundaries and the aggregated 
information associated with each bucket lead to different families of histograms. Below 
we review the most common types of histograms, and then we explain how histograms 
are used to estimate cardinality values of input queries.

2.2.1.1 Unidimensional Histograms

Unidimensional histograms (i.e., histograms that are built over a single attribute) are 
the most widely used statistics in commercial database systems. One reason is that 
unidimensional histograms can be built and maintained efficiently by using sampling 
techniques [CMN98]. Another reason is that the number of histograms that might be 
useful for a given query is manageable, and therefore all possible histograms can be 
exploited simultaneously during optimization with little overhead. We now summarize 
the most common types of unidimensional histograms and review the specific procedure
they use to select bucket boundaries (Figure 2.4 shows instances of such histograms for a simple data distribution).

- *EquiWidth* [Koo80]: The data domain is divided into subranges of equal size. *EquiWidth* were the first proposed histograms and are very simple to understand. However, the rigid procedure to select buckets boundaries makes these histograms not so accurate in practice for cardinality estimation.

- *EquiDepth* [PSC84]: The data domain is divided into subranges such that the total number of tuples \( f_i \) in each range is roughly the same, and so bucket boundaries correspond to common “quantiles.” *EquiDepth* histograms thus take into account the actual data distribution to select bucket boundaries, and for that reason result in better cardinality estimates than those derived using *EquiWidth* histograms.

- *MaxDi* [PIHS96]: Bucket boundaries are placed between pairs of consecutive values that have the largest “gap” in terms of frequency. The procedure for selecting bucket boundaries is a heuristic designed to minimize the variance of tuple frequencies inside buckets.

- *End-Biased* [PIHS96]: Separate counts are maintained for a small number of very frequent values, and the remaining values are modelled using an *EquiDepth* histogram. *End-Biased* histograms are motivated by the fact that data sets are often skewed, with a small number of very frequent values.

To conclude this section, we mention other less common histogram techniques. Reference [KW99] combines histograms and parametric curve fitting, and proposes spline-based histograms in which each bucket is associated with a linear spline instead of with a single frequency value. Both the spline parameters and the bucket boundaries are chosen to fit observed data distributions. Reference [PIHS96] introduces *V-optimal* histograms, which minimize the variance of the overall frequency approximation. A dynamic programming algorithm is presented in [JKM+98] for building *V-optimal* histograms in \( O(N^2b) \) time,
Figure 2.4: Histograms approximate the value distribution of columns in a database.

where \( N \) is the number of tuples in the data set and \( b \) is the number of buckets. Finally, references [GMP97, DIR00] present reorganization strategies for unidimensional histograms. In these references, histograms are refined as the corresponding base tables change, and are periodically reorganized if the number of data updates or some other inaccuracy threshold is exceeded.
2.2.1.2 Multidimensional Histograms

Unidimensional histograms provide information on a single column, but fail to model the correlation between columns. In fact, correlations are only captured by summarizing the joint distribution of multiple columns. Multidimensional histograms generalize the concepts introduced in the previous section to multiple dimensions.

Similar to the unidimensional case, a multidimensional histogram partitions the data domain in a set of (hyper-) rectangular buckets that cover all the points in the domain, and assigns to each bucket some aggregated information (usually the total number of tuples and the number of distinct values enclosed in the bucket, as in the unidimensional case). The choice of rectangular buckets is justified by two main reasons. First, rectangular buckets make it easy and efficient to intersect each bucket and a given range query to estimate cardinality values. Second, rectangular buckets can be represented concisely, which allows a large number of buckets to be stored using the given budget constraints. However, partitioning multidimensional spaces is challenging, and simple generalizations of techniques used for a single column are often not adequate. In the rest of this section we review proposed multidimensional histograms.

A multidimensional version of the EquiDepth histogram, presented in [MD88], recursively partitions the data domain, one dimension at a time, into buckets enclosing the same number of tuples. Reference [PI97] introduces MHist, a generalization of MaxDiff histograms that iteratively partitions the data domain using a greedy procedure. At each step, MHist analyzes unidimensional projections of the data set and identifies the bucket in most need of partitioning (such a bucket has the largest “gap” between two consecutive values along one dimension). Using this information, MHist iteratively splits buckets until it reaches the desired number of buckets. Reference [GKTD00] presents GenHist histograms, which allow unrestricted overlap among buckets. If more than two buckets overlap, the density of tuples in their intersection is approximated as the sum of the data densities of the overlapping buckets. A tuple that lies in the intersection of many buckets is counted in only one of them (chosen probabilistically). The technique constructs pro-
gressively coarser grids over the data set, convert the densest cells into buckets, and then removes a percentage of tuples in those cells to make the resulting distribution smoother.

All the histogram techniques above are static: after histograms are built, their buckets and frequencies remain fixed regardless of any changes in the data distribution. Reference [AC99] presents STGrid, the first multidimensional histogram that uses query feedback to refine buckets histograms. An STGrid histogram partitions the data domain into disjoint buckets that form a grid. Bucket boundaries along each dimension are defined using MaxDiff histograms, and their frequencies are refined using query feedback. After a predetermined number of queries, the histogram is restructured by merging and splitting rows of buckets at a time, to preserve the grid structure. For that reason, STGrid histograms generally contains many not-so-useful buckets. To avoid the poor bucket layout problems of STGrid histograms and still use query workloads to refine histograms, we introduced STHoles [BCG01a], a new partitioning scheme for building multidimensional histograms that allows buckets to overlap. Specifically, inclusion relationships are allowed between buckets (some buckets can be completely included inside others). Therefore, we implicitly relax the requirement that regions be rectangular while keeping rectangular bucket structures. By allowing bucket nesting, the resulting histograms can model complex shapes (not restricted to rectangles anymore); by restricting the way in which buckets may overlap, the resulting histograms can be efficiently created and updated incrementally by using workload information. In contrast to previous multidimensional histogram techniques that use unidimensional projections of the data set for bucket creation, STHoles exploits query feedback in a truly multidimensional way to improve the quality of the resulting histograms. Our extensive experiments demonstrate that STHoles histograms consistently produce good cardinality estimates across synthetic and real-world data sets and across query workloads, and, in many cases, outperform the best multidimensional histogram techniques that require access to and processing of the full data during histogram construction. See Chapter 5 for a more detailed description and evaluation of STHoles histograms.
Figure 2.5: Multidimensional histograms approximate joint distributions of values.

To summarize the different partitioning strategies discussed above, Figure 2.5(a) shows a synthetically generated data set consisting of multiple overlapping Gaussian distributions, while Figures 2.5(b-f) present instances of the multidimensional histograms described in this section. All histograms in the figures use the same amount of memory, which in turn results in different numbers of buckets allocated to each histogram, since the amount of information needed to describe each bucket is different across histograms.

We conclude this section by noting that multidimensional histograms are not widely used in commercial database systems yet. One reason is that the number of candidate histograms is too large even for a moderate number of dimensions. Also, it is not always clear how to select useful sets of columns over which histograms are built. (Building a single multidimensional histogram per table using all columns does not scale, since histograms become too inaccurate beyond 4 or 5 dimensions). Finally, multidimensional histogram techniques are relatively new, and their interaction with other cost estimation components is not fully understood yet.
2.2.2 Cardinality Estimation using Histograms

In this section we review how current systems typically use histograms to estimate query cardinality values. We focus on unidimensional histograms since, as explained earlier, they are used almost exclusively in commercial databases. Many ideas, however, can be easily extended to multidimensional histograms as well.

Recall from the previous section that each bucket $b_i$ typically has two associated values: $f_i$ (the total number of tuples in the bucket) and $dv_i$ (the total number of distinct values in the bucket). The fundamental assumption when using histograms for cardinality estimation is that the distribution of tuples inside each bucket is uniform. In particular, we use the uniform spread model inside buckets, which states that each bucket $b_i$ corresponds to $dv_i$ equidistant “groups” of $f_i/dv_i$ tuples each. We usually refer to the density of a bucket as $\delta_i = f_i/dv_i$, or the number of tuples per distinct value (assuming uniformity) that are represented in the bucket.

![Figure 2.6: Uniform spread and frequency assumption for histogram buckets.](image)

**Example 3** Consider the data distribution of Figure 2.6(a), and suppose that we build the two-bucket histogram of Figure 2.6(b) over this data set. In this case, we view bucket $b_1$ as corresponding to $dv_1 = 2$ groups of $f_1/dv_1 = 5$ tuples each. Similarly, we view the second bucket as corresponding to $dv_2 = 3$ groups of $f_2/dv_2 = 4$ tuples each. The density of buckets $b_1$ and $b_2$ is then 5 and 4, respectively. Figure 2.6(c) shows the approximate table consistent with this bucket interpretation.


CHAPTER 2. OPTIMIZATION IN RELATIONAL DATABASE SYSTEMS

Using this basic assumption for histogram buckets, in the rest of this section we discuss procedures to estimate the cardinality of queries using histograms.

2.2.2.1 Selection Queries

The uniformity assumption for histogram buckets suggests a natural interpolation-based procedure to estimate the cardinality of selection queries. This estimation is particularly simple for range predicates. Consider query \( q = \sigma_{5 \leq R.a \leq 30}(R) \) and histogram \( H(R.a) \). To estimate the cardinality of \( q \), we consider each histogram bucket that “intersects” with the range predicate, estimate the contribution in cardinality for each bucket, and finally aggregate all partial results. This simple procedure is illustrated in the following example.

Example 4 Consider the four-bucket histogram \( H(R.a) \) of Figure 2.7 and the range predicate \( p = 5 \leq R.a \leq 30 \). Since \( p \) completely includes bucket \( b_2 \), we know that the 50 tuples in \( b_2 \) must satisfy \( p \). Also, \( p \) is incompatible with bucket \( b_4 \), so no tuple in \( b_4 \) satisfies \( p \). Finally, \( p \) partially overlaps with buckets \( b_1 \) and \( b_3 \) (in particular, \( p \) “covers” 50% of \( b_1 \) and 33% of \( b_3 \)'s uniformly spread tuples). We then estimate that 100/2 tuples in \( b_1 \) and 80/3 tuples in \( b_3 \) satisfy \( p \). In summary, the number of tuples satisfying predicate \( p = 5 \leq R.a \leq 25 \) is estimated as \( 100/2 + 50 + 80/3 = 126.67 \).

![Figure 2.7: Cardinality estimation for range queries using histograms.](image-url)
Other types of selection predicates are estimated in similar ways. For instance, open-ended range predicates (e.g., \( R.a \leq 20 \)) are transformed into range queries with extreme values (e.g., \( -\infty < R.a \leq 20 \)). The special case of a point selection such as \( R.a = 20 \) is treated slightly differently. In this case, the estimated cardinality is defined as the density of the bucket that covers the value 20 (for instance, the cardinality of predicate \( R.a = 20 \) using the histogram in Figure 2.7 is estimated as \( f_2/dv_2 = 6.25 \)). This procedure slightly overestimates the cardinality of \( R.a = 20 \) returning an estimate that is usually different from that of \( 20 \leq R.a < 21 \). If \( R.a \) is defined over integer values, though, the cardinality of \( R.a = 20 \) can be estimated by using that of \( 20 \leq R.a < 21 \) instead. Also, if the histogram is end-biased and the point selection value is one of the distinguished values for which the exact cardinality is stored, such precise frequency is returned. Finally, some subtleties arise when the query boundaries agree with the bucket boundaries, or when using open (\( \leq, \geq \)) versus closed (\( <, > \)) ranges in discrete domains. Those borderline cases need to be handled carefully, and the details are specific to each implementation.

Selection queries may contain multiple predicates on different attributes of a table. Consider for instance query \( q = \sigma_{R.a > 10 \land R.b < 20}(R) \). If histograms \( H(R.a) \) and \( H(R.b) \) are available, we first approximate the cardinality of each predicate in isolation using the corresponding histogram. Suppose that \( C_a \) and \( C_b \) are the cardinality estimates for \( R.a > 10 \) and \( R.b < 20 \), respectively. The cardinality of the whole query is then estimated as \( C_a \cdot C_b/|R| \), where \(|R|\) is the number of tuples in \( R \). Therefore, if \( S_a = C_a/|R| \) is the selectivity\(^1\) of \( R.a > 10 \) and \( S_b = C_b/|R| \) is the selectivity for \( R.b < 20 \), the combined selectivity is estimated, assuming independence between predicates, as \( S_a \cdot S_b \). If a two-dimensional histogram \( H(R.a, R.b) \) is available, the estimation of both predicates can be done simultaneously using a procedure similar to that of Figure 2.7, without assuming independence. Finally, for complex Boolean predicates including or's and not's, logical identities are used. For instance, the cardinality of \( \sigma_{R.a < 10 \lor R.b > 20}(R) \) is equivalent to \(|R|\) minus the cardinality of \( \sigma_{R.a \geq 10 \land R.b \leq 20}(R) \), which is evaluated as explained before.

\(^1\)The selectivity of a given predicate \( p \) is defined as the fraction of tuples that satisfy \( p \).
2.2.2.2 Join Queries

Histograms can also be used to estimate the cardinality of join predicates. Consider for instance query $R \bowtie_{x=y} S$. Figure 2.8(a) shows histograms $H(R.x)$ and $H(S.y)$ over $R.x$ and $S.y$, respectively, where each histogram bucket is delimited by square brackets. The procedure to estimate the cardinality of the join predicate using these histograms consists of the following three steps:

1. We align the histogram buckets (possibly splitting some buckets) so that each bucket boundary in one histogram has a counterpart in the other histogram. For instance, buckets $b_2$ and $b'_2$ in the figure share the same left boundary. However, bucket $b_2$ spans beyond bucket $b'_2$’s right boundary. Therefore, we split bucket $b_2$ into two sub-buckets. The left sub-bucket boundary agrees with that of bucket $b'_2$. The right sub-bucket starts at the same position as bucket $b'_3$ but ends before $b'_3$ does. Then, bucket $b'_3$ is split in the same way, and this procedure continues until all original buckets are aligned (see Step 1 in Figure 2.8). This step is guaranteed to at most double the total number of buckets in both histograms.

2. We analyze each pair of aligned buckets and do per bucket estimation of join sizes. There is no well-founded approach towards doing it and here we sketch the most common alternative. First, we use the containment assumption [SAC+79] and therefore consider that each group of distinct valued tuples belonging to the bucket with the minimal number of different values joins with some group of tuples in the other bucket (the containment assumption is inspired by the common case of foreign-key joins). For instance, in Step 2 of Figure 2.8, the three groups of tuples from the upper bucket are assumed to match with three of the five groups of tuples in the lower bucket. We can model the result of joining the pair of buckets as a new bucket with three distinct values and density $40 = 2 \cdot 20$. That is, each distinct value in the resulting bucket represents 40 tuples, which is the product of the original bucket densities. Therefore, the frequency of the new bucket is $120 = 3 \cdot 40$. 
3. After applying the same procedure to each pair of aligned buckets, we add the partial frequencies from each resulting bucket and obtain the estimated cardinality for the original join query.
We note that in Step 2 the new temporary histogram may not be necessarily materialized. In effect, if this intermediate histogram is no further needed during cardinality estimation, Steps 2 and 3 above are interleaved without materializing temporary buckets.

The procedure described above can be slightly modified to handle other kinds of joins. Consider query $q = R \bowtie_{R.x \leq S.y} S$ and histograms $H(R.x)$ and $H(S.y)$. Then, for each bucket $b$ from $H(S.y)$, Step 2 needs to consider all buckets $b'$ of $H(R.x)$ to the left of the one aligned with $b$ (inclusive). Other types of joins, such as semi-joins or outer-joins can also be handled by slightly modifying Step 2 in the procedure above.

### 2.2.2.3 Select-Project-Join Queries

The techniques discussed in the previous section can be applied to queries with simple structure (e.g., pure joins or range queries). When considering arbitrary SPJ queries, we face the additional challenge that cardinality estimation requires propagating statistics through predicates. Consider the query:

```sql
SELECT * FROM R,S
WHERE R.x=S.y AND S.a<10
```

and assume that histograms $H(R.x)$, $H(S.y)$, and $H(S.a)$ are available. There are two alternative ways to estimate the cardinality of the whole expression, associated with the equivalent expressions $\sigma_{S.a<10}(R \bowtie_{R.x=S.y} S)$ and $R \bowtie_{R.x=S.y}(\sigma_{S.a<10}S)$, respectively. One alternative (Figure 2.9(a)) uses histograms $H(R.x)$ and $H(S.y)$ to estimate the cardinality of $R \bowtie_{R.x=S.y} S$ ignoring the predicate $S.a < 10$. Then, the histogram for $S.a$ is propagated\(^2\) through the join upwards in the tree assuming independence between $S.y$ and $S.a$. The propagated histogram is then used to estimate the cardinality of $S.a < 10$ over the result from $R \bowtie_{R.x=S.y} S$, which is the cardinality of $\sigma_{S.a<10}(R \bowtie_{R.x=S.y} S)$. The other possibility (Figure 2.9(b)) first uses the histogram $H(S.a)$ to estimate the cardinality

\(^2\)The histogram propagation step just scales the bucket frequencies so that they reflect the new cardinality information. In this case, the frequency values for histogram $S.a$ are scaled so that the sum of all frequencies in the histogram equals the estimated number of tuples in $R \bowtie_{R.x=S.y} S$.  

of $\sigma_{S.a<10}(S)$. Then, assuming independence between $S.y$ and $S.a$, the histogram for $S.y$ is propagated through the selection operator and used together with histogram $H(R.x)$ to estimate the cardinality of $R \bowtie_{R.x=S.y}(\sigma_{S.a<10}(S))$.

The procedure of estimating partial cardinality values and propagating the remaining histograms can be generalized to estimate the cardinality of arbitrary SPJ queries.

### 2.2.2.4 Queries with Group-by Clauses

In this section we describe some techniques to estimate the cardinality of SPJ queries with group-by clauses. Consider as an example the following query:

```sql
SELECT R.a, sum(R.c)
FROM R
GROUP BY R.a
```

The cardinality of this query is equal to the number of groups in the output (i.e., the number of distinct values of $R.a$ in $R$). This value can be approximated using histogram $H(R.a)$. Recall that each histogram bucket $b_i$ contains a value $dv_i$ that represents the number of distinct values in $b_i$. Therefore, the cardinality of the query above is estimated as the sum, over all buckets, of the number of distinct values per bucket (i.e., $\sum dv_i$).

When more than one attribute is mentioned in the group-by clause, we can either use a multidimensional histogram (if available), or we can alternatively obtain the number of distinct values for each attribute individually using unidimensional histograms, and then combine the results using some simplifying assumption. For instance, consider the case in
which we add $R.b$ to the *group-by* clause in the query above, so that it becomes `GROUP BY R.a, R.b`. If we estimate that $D_a$ and $D_b$ are the number of distinct values of $R.a$ and $R.b$, respectively, we can estimate the cardinality of the whole query as $\min(|R|, D_a \cdot D_b)$. That is, we assume that the distribution of attributes $R.a$ and $R.b$ are independent, and therefore each combination of values of $R.a$ and $R.b$ is present in $R$ (we cap the resulting value by the absolute upper bound of $|R|$ itself). This is usually an overestimation, and some alternative procedures are used in specific systems. For instance, some database systems explicitly maintain the number of distinct values for some combination of attributes (usually prefixes of multi-column indexes) and therefore these cardinality values can be estimated directly.

To conclude this section, we mention that in presence of more complex queries, the techniques discussed above are combined using independence or other assumptions, but the details vary considerably across specific implementations.

### 2.2.3 Cardinality Estimation in the Absence of Histograms

Up to now, we discussed various techniques that approximate query cardinality values, assuming that the necessary histograms are available. Whenever some of the required histograms are not present, the optimizer uses alternative (less accurate) procedures to estimate cardinality values. For instance, if the database records the total number of tuples in each table, together with the minimum and maximum values for each column in each table, range queries can be estimated assuming uniformity by using interpolation (which is equivalent to using a single-bucket histogram built with the available information). In the extreme situation that no statistical information is available, optimizers assign fixed “reduction factors” to each predicate, and then combine partial results assuming independence. For instance, to estimate the cardinality of $\sigma_{R.a < 10}(R)$ in the absence of statistical information, the optimizer uses the fixed value $1/3$ and estimates the cardinality as $|R|/3$. Another example of a reduction factor is $1/10$ for equality-join predicates such as $R \bowtie_{R.x = S.y} S$. 
We note that some current optimizers, such as that of Microsoft SQL Server, often avoid relying on such “magic numbers” by fully automating the creation of histograms. In these systems, optimization is paused whenever a referenced column has no associated histogram, an appropriate histogram is created efficiently by using sampling [CMN98], and then optimization is resumed. For that reason, the first queries posed to the system might experience longer response times than later queries because of the additional work of creating histograms. Generally, the system quickly stabilizes and the database administrator is freed from choosing which histograms to build and when to do so.

2.3 Conclusion

In this chapter we reviewed the basic architecture of state-of-the-art query optimizers, and discussed the steps involved in obtaining an efficient execution plan for input queries. We identified the cost estimation module (and, specifically, the cardinality estimation routines) as a critical component in the optimizer that can have a large impact on the quality of the resulting query plans. We then discussed histograms as the main source of statistical information in a database system, and showed how they can be used to estimate cardinality values of intermediate results produced by the sub-plans that are considered during optimization. We also reviewed some of the assumptions that are used during cardinality estimation of complex queries. In the next chapter we critically examine such assumptions and argue that—in many scenarios—they can lead to inaccurate cardinality estimates, which in turn can cause the optimizer to choose low-quality execution plans. We then introduce SITs to diminish propagation of errors in complex query plans, and discuss the main challenges that we need to address before SITs can be incorporated in real relational database systems.
Chapter 3

SITs: Statistics on Query Expressions

In the previous chapter we discussed different techniques that exploit base-table statistics to approximate the cardinality of input query plans that are generated during optimization. We also showed that many of those procedures rely on (sometimes natural) simplifying assumptions. In this chapter we start by taking a closer look at those simplifying assumptions and illustrate how cardinality estimates can become very inaccurate if the underlying assumptions do not hold (Section 3.1). Unfortunately, this is the common scenario, so usually the optimizers rely on wrong cardinality estimates and potentially produce low-quality execution plans. To overcome this limitation, in Section 3.2 we introduce SITs, which can be regarded as additional statistical information for the optimizer. As we will see, SITs can significantly reduce the propagation of errors during cardinality estimation and help the optimizer obtain significantly better-quality execution plans. However, some important challenges need to be addressed to make SITs an effective tool in modern database systems. We briefly discuss such challenges in Section 3.2.2 and defer a detailed analysis of each problem to Chapters 4 through 6.
3.1 Simplifying Assumptions in Cardinality Estimation

We now discuss the main simplifying assumptions that query optimizers typically make to estimate the cardinality of a query, and show how those assumptions can affect the quality of the estimate. Consider the query below:

```
SELECT * FROM R, S
WHERE R.x=S.y and S.a<10
```

and recall one way to estimate its cardinality assuming that all required histograms are available (Section 2.2.2.3):

1. Use histograms \( H(R.x) \) and \( H(S.y) \) to approximate the cardinality of the join predicate \( R \bowtie_{R.x=S.y} S \).

2. Propagate histogram \( H(S.a) \) so that its total number of tuples equals the cardinality estimation of the previous step.

3. Use the propagated histogram to approximate the range query \( S.a < 10 \).

The procedure above relies on three simplifying assumptions, namely the *independence* assumption between attributes, the *containment* assumption for joins, and *sampling* assumption for base-table histograms (see Figure 3.1). We discuss each assumption below.

![Figure 3.1: Assumptions to estimate the cardinality of an SPJ query.](image-url)
3.1.1 Independence Assumption

When propagating histogram $H(S.a)$ through the predicate $R \bowtie_{x=y} S$ in Step 2 above, each of its bucket frequencies is uniformly scaled. In the propagated histogram, the sum of all its bucket frequencies equals the estimated cardinality of $R \bowtie_{x=y} S$ in Step 1. Implicit in this procedure is the assumption that the distributions of attributes in $R$ and $S$ are independent, and therefore the same scaling factor is used for all buckets of the original histogram. Unfortunately, in many cases there are complex correlations between attributes, so relying on the independence assumption can result in arbitrarily large estimation errors. In some cases (e.g., queries with multiple filter predicates over a single table), multidimensional histograms, if available, can estimate cardinality values without relying on the independence assumption. However, in many other situations multidimensional histograms are not appropriate. For instance, correlations between attributes in different tables that are subsequently joined cannot be modelled by traditional multidimensional histograms. In fact, the independence assumption is the main source of errors during cardinality estimation, and its pervasive usage hinders the ability of the optimizer to obtain efficient execution plans. The following example illustrates how using the independence assumption can result in a very inaccurate cardinality estimation of the query in Figure 3.1 for a specific database instance.

Example 5 Consider the query shown in Figure 3.1 and suppose that Figure 3.2(a) shows the actual contents of table $S$. Figure 3.2(b), in turn, shows histogram $H(S.a)$ (the histogram in this case is optimal, since it consists of five buckets, each one covering one of the five different values of $S.a$). Now suppose that table $R$ is very skewed towards value $v_1$, so most of the tuples in $R \bowtie_{R.x=S.y} S$ have value $v_1$ in the joined column (Figure 3.2(c)). The estimated cardinality of the join using histograms is, in the best case, equal to 100 (it can be slightly different due to other assumptions made while estimating the join using histograms). Therefore, to propagate histogram $H(S.a)$ assuming independence, we multiply each bucket frequency by $100/5 = 20$, which results in the propagated histogram of Figure 3.2(d). A range estimation $S.a < 10$ over that histogram returns 20, which is the
CHAPTER 3. SITS: STATISTICS ON QUERY EXPRESSIONS

(a) Table S.

<table>
<thead>
<tr>
<th>a</th>
<th>v1</th>
<th>v2</th>
<th>v3</th>
<th>v4</th>
<th>v5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>5</td>
<td>10</td>
<td>15</td>
<td>20</td>
<td>25</td>
</tr>
</tbody>
</table>

(b) Histogram $H(S.a)$.

(c) Joining $R$ and $S$.

<table>
<thead>
<tr>
<th>x</th>
<th>y</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v1</td>
<td>5</td>
</tr>
<tr>
<td>v2</td>
<td>v2</td>
<td>10</td>
</tr>
<tr>
<td>v3</td>
<td>v3</td>
<td>15</td>
</tr>
<tr>
<td>v4</td>
<td>v4</td>
<td>20</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>y</th>
<th>a</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1</td>
<td>v1</td>
</tr>
<tr>
<td>v2</td>
<td>v2</td>
</tr>
<tr>
<td>v3</td>
<td>v3</td>
</tr>
<tr>
<td>v4</td>
<td>v4</td>
</tr>
</tbody>
</table>

(d) Propagated histogram on $S.a$ when assuming independence with the join predicate.

(e) Actual result.

(f) Propagated histogram over the actual result.

Figure 3.2: Effect of the independence assumptions during cardinality estimation.

estimated cardinality of the query. However, we can see in Figure 3.2(e) that the actual cardinality of the query is 97, which is almost 5 times larger than the approximation that relies on independence. In fact, the correct propagated histogram, without assuming independence, is shown in Figure 3.2(f), and its distribution is considerably different from that of the histogram in Figure 3.2(d). Unfortunately, the histogram in Figure 3.2(f) cannot be derived from the original unidimensional base-table histograms without looking at the actual joint data distribution.
Join predicates are not the only case that relies on independence between attributes. On the contrary, this assumption is pervasive during cardinality estimation. For instance, to estimate the cardinality of \( \sigma_{R.a<10 \land R.b>20}(R) \) using unidimensional histograms \( H(R.a) \) and \( H(R.b) \), the corresponding cardinality values are combined using independence. This scenario can also be viewed as propagating histogram \( H(R.a) \) through the operator \( R.b > 20 \). Alternatively, we can consider the equivalent query plan \( \sigma_{R.a<10}(\sigma_{R.b>20}(R)) \) and propagate histogram \( H(R.b) \) through the operator \( R.a < 10 \). Finally, we stress that for complex queries, the independence assumption further reduces the accuracy of the estimations, since errors propagate through the query plan along with the histograms. After a few operators, the resulting estimates can be off by orders of magnitude [IC91].

### 3.1.2 Containment Assumption

The independence assumption is one of the most important sources of error during cardinality estimation, but it is far from being the only source. Some operators rely on other assumptions that can result in further inaccuracies. As a notable example, the estimation of a join predicate using histograms typically relies on the containment assumption for histogram buckets, which models the join as if it were between foreign and primary keys (see Section 2.2.2.2). This assumption is clearly satisfied for foreign-key joins, but it might be violated for other types of joins. However, it is also a good approximation in many other cases, since we would intuitively expect that if some bucket \( b \) has many different values, then a given value from the corresponding bucket in the other histogram has a good chance of matching some value in \( b \). For that reason, the containment assumption is not considered as harmful as the independence assumption in real situations, but it certainly plays a role in introducing new (or amplifying previous) estimation errors.

### 3.1.3 Sampling Assumption

For efficiency, some database systems use adaptive random sampling to build approximate histograms when it is not possible to derive them exactly from materialized indexes. It
is shown in [CMN98] that the approximate histograms are good estimators for frequency distributions. Thus, histograms obtained by using sampling can accurately estimate the cardinality of range queries. However, approximating the number of distinct values inside buckets using sampling is provably difficult [CCMN00]. If the number of distinct values within buckets is not correct, the quality of cardinality estimates for join or group-by queries can be compromised. The sampling assumption states that the number of distinct values in each bucket predicted by using adaptive sampling is a good estimator of the actual values. In practical situations, the sampling assumption usually amplifies errors during cardinality estimation but it is not as harmful as the independence assumption.

3.1.4 Other Assumptions

We conclude this section by mentioning that other simplifying assumptions are used in specific systems. For instance, the preservation of value sets [GMUW00] states that when joining tables $R$ and $S$, any attribute from $R$ or $S$ that is not a join attribute does not lose values from its set of possible values (this assumption is used to estimate the cardinality of multi-way joins). Another example is the nesting assumption for queries with multiple attributes in the group-by clause, which states that the number of distinct values considering attributes $\{a_1, \ldots, a_n\}$ is equal to the minimum number of distinct values considering each $a_i$ individually.

In general, all the simplifying assumptions discussed in this chapter can result in significantly inaccurate cardinality estimations for complex queries. However, in the absence of additional information, all we can do is to review such assumptions and tweak them hoping to obtain better cardinality estimates in more scenarios. We next take a different approach and introduce SITs, which can be seen as additional statistical information available to the optimizer during cardinality estimation. As we will see, by using SITs carefully we can avoid relying on many of the simplifying assumptions described earlier, hence reducing errors during cardinality estimation.
Figure 3.3: Using statistics over views to improve cardinality estimation.

3.2 SITs: Avoiding Error Propagation

We begin this section with a simple example to motivate the introduction of SITs. Consider Figure 3.3(a), which shows again the query used in Section 3.1 to analyze different simplifying assumptions. Now suppose that we evaluate $R \bowtie_{R.x=S.y} S$ and materialize the result as a new table $RS$. We can then rewrite the query in Figure 3.3(a) as in Figure 3.3(b). Both queries are conceptually equivalent and produce the same result. Interestingly, the transformed query of Figure 3.3(b) has an advantage from a cardinality estimation perspective: since $RS$ is a regular table, we can create a traditional histogram over column $RS.a$ (i.e., $H(RS.a)$). Now, we can approximate the cardinality of the original query using a simple range estimation over histogram $H(RS.a)$. Thus we avoid relying on procedures that involve three histograms ($H(R.x)$, $H(S.y)$, and $H(S.a)$) and using multiple simplifying assumptions\(^1\). For complex query plans, having statistics on subexpressions of the query is highly beneficial, since it limits the propagation of errors through sequences of operators. We now formalize the notion of SITs.

**Definition 2** Let $Q$ be an SQL query and $A$ be some of the table attributes in the answer to $Q$. For a given histogram construction strategy, $\text{SIT}(A|Q)$ is defined as the histogram over $A$ for the $Q$ result. We refer to $Q$ as the generating query expression of $\text{SIT}(A|Q)$.

\(^1\)Note that table $RS$ does not need to be actually materialized. In fact, we only take advantage of histogram $H(RS.a)$. 

SITs and traditional base-table histograms have the same internal structure and operations. In fact, we can regard a base-table histogram $H(R.a)$ as $\text{SIT}(R.a|Q_0)$, where $Q_0$ is the trivial query $\text{SELECT } * \text{ FROM } R$.

**Example 6** Consider query $Q$, defined as

```sql
SELECT R.a FROM R,S
WHERE R.x=S.y
```

A MaxDiff $\text{SIT}(R.a|Q)$ is a histogram over attribute $R.a$ built on $Q$’s result. $\text{SIT}(R.a|Q)$ consists of buckets, just as any traditional MaxDiff histogram does, and can be used for cardinality estimation using the techniques described in Section 2.2.2.

SITs and base-table multidimensional histograms are not incompatible statistics. In fact, a multidimensional histogram is just a SIT over multiple columns defined over a trivial generating query. For some queries (e.g., single-table queries with multiple filter predicates), base-table multidimensional histograms are the best alternative to estimate cardinality values. Analogously, for the SPJ query of Figure 3.3, a single-column $\text{SIT}(S.a|R \bowtie S)$ is the most appropriate alternative. Finally, other queries can exploit multidimensional SITs with complex generating queries. SITs can be regarded as unifying single- and multi-column histograms defined over either base tables or generating query expressions. In this thesis, though, we mostly focus on single-column SITs over arbitrary generating query expressions, since those statistics can be easily incorporated in current optimizers.

Finally, we note that SITs can be further generalized to other statistical summaries. For example, we could allow $\text{SIT}(R.a|Q)$ to be a wavelet over attribute $R.a$ built for $Q$’s result. For concreteness, in this thesis we follow Definition 2 and assume that SITs are histograms, but many of our results apply to other statistical summaries as well.
3.2.1 A Validating Experiment

Before adopting SITs as a viable approach to reduce errors in cardinality estimation, we need to convince ourselves that SITs can positively impact query processing. In this section we conduct an experiment to show the potential of using SITs during optimization. For that purpose, we use the popular TPC-H benchmark schema (see [TPC02] for a detailed description of TPC-H). Unfortunately, one of the requirements of the benchmark is that data be generated from a uniform distribution. Likewise, there is a constraint in the number of matches per tuple in all foreign-key joins (e.g., each order tuple has associated \( n \) lineitem tuples, where \( n \) is a random integer between one and seven). To model real scenarios, we extend the TPC-H generation program to produce tables with varying degree of skew. In particular, the generator produces data for each column in the schema from a Zipfian distribution [Zip49], further extending the modifications proposed in [CN00]. Zipfian distributions follow a simple power law and are commonly observed in many kinds of real-world phenomena. We also used Zipfian distributions to model foreign-key joins, so for instance the number of tuples in lineitem that join with each tuple in orders is not uniform.

We generated the TPC-H data sets using a skew factor \( z = 1 \) and a resulting database size of 1 GB, and used the following SQL query, which asks for information about the most expensive orders (i.e., those with a total price greater than 1,000,000):

\[
\begin{align*}
\text{SELECT} & \quad * \\
\text{FROM} & \quad \text{lineitem, orders, part, supplier} \\
\text{WHERE} & \quad l\text{.orderkey} = o\text{.orderkey} \quad \text{and} \\
& \quad l\text{.partkey} = p\text{.partkey} \quad \text{and} \\
& \quad l\text{.suppkey} = s\text{.suppkey} \quad \text{and} \\
& \quad o\text{.totalprice} > 1000000
\end{align*}
\]

In our database, the cardinality of \( \sigma_{o\text{.totalprice}>1,000,000} \) (orders) is 120: 120 out of 750,000 tuples in orders satisfy the filter condition (i.e., the selectivity of this selection condition is lower than 0.02\%). These 120 tuples join with a very large number of tuples
in `lineitem`, which is precisely why these orders are so expensive. In fact, the cardinality of \( \sigma_{o_{\text{totalprice}}>1,000,000} (\text{orders} \bowtie \text{lineitem}) \) is 971,851 out of 2,943,815 tuples (i.e., the selectivity is around 33%). Clearly, if we simply propagate the histogram for \( o_{\text{totalprice}} \) through the join \( \text{lineitem} \bowtie \text{orders} \), we incur large estimation errors, which in turn affect the optimizer’s choice of an execution plan.

We first optimized the query above using Microsoft SQL Server’s optimizer, and obtained the query execution plan of Figure 3.4(a). In this case, the optimizer estimates that the result size of the sub-query \( \text{lineitem} \bowtie \sigma_{o_{\text{totalprice}}>1,000,000} (\text{orders}) \) is small (only 713 tuples), so it chooses to sort this intermediate result before passing it to the next join with `supplier`. Since the estimated intermediate result is still small, a nested loop join is used with `part` to obtain the final result.

![Figure 3.4: Query execution plans for different sets of available SITs.](image)

(a) No SITs are available.  (b) All SITs are available.

To establish the potential benefit of using SITs, we intercept each call to the cardinality estimation module inside the optimizer and modify each original estimated cardinality with its exact value. This simple modification represents the best possible scenario in which the optimizer exploits an available SIT that matches the input plan exactly,
and such SIT is as accurate as possible, returning the exact cardinality estimation. We optimized the same query in this way and obtained the alternative query plan of Figure 3.4(b). In this case, the optimizer accurately estimates that the number of tuples in \texttt{lineitem \ \bowtie \ \orders} is large (970,627 tuples) and chooses a different set of operators. In particular, the expensive sort operation is removed and some joins are replaced with more efficient hash joins. In some cases, the inner/outer role of the tables is additionally reversed.

Finally, we evaluated the two query execution plans of Figure 3.4 and measured both the CPU and I/O execution times for each alternative. Figure 3.5 reports the results averaged over five independent executions. The elapsed time of the plan obtained when no SITs are considered is 419 seconds. In contrast, the plan produced when all possible SITs are available to the optimizer required just 23 seconds (i.e., less than 6% of the time spent by the original plan). Clearly, in this ideal scenario the presence of SITs dramatically reduces the execution cost of the input query, showing the potential benefit of our approach.

![Figure 3.5: By using SITs, execution times can be drastically reduced.](image)

**3.2.2 Exploiting SITs: Challenges**

Despite the conceptual simplicity of SITs, significant challenges need to be addressed before they can be effectively incorporated in existing RDBMSs.
First, we must adapt query optimizers to exploit the additional statistical information provided by SITs. This is not an easy task, since multiple SITs can be useful to only portions of the input query, and some SITs can be mutually incompatible. For instance, consider the query in the previous section and the following SITs:

\[ S_1 = \text{SIT}(o_{\text{totalprice}} \bowtie \text{orders} \bowtie \text{lineitem} \bowtie \text{part}) \]
\[ S_2 = \text{SIT}(o_{\text{totalprice}} \bowtie \text{orders} \bowtie \text{lineitem} \bowtie \text{supplier}) \]

At first glance, it might seem that SITs \( S_1 \) and \( S_2 \) can only be exploited to estimate the cardinality of the sub-queries \( \sigma_{o_{\text{totalprice}}} > 1,000,000 \) \((\text{orders} \bowtie \text{lineitem} \bowtie \text{part})\) and \( \sigma_{o_{\text{totalprice}}} > 1,000,000 \) \((\text{orders} \bowtie \text{lineitem} \bowtie \text{supplier})\), respectively. As we will see, we can also use either \( S_1 \) or \( S_2 \) (but not both) to estimate the cardinality of the original query \( \sigma_{o_{\text{totalprice}}} > 1,000,000 \) \((\text{orders} \bowtie \text{lineitem} \bowtie \text{part} \bowtie \text{supplier})\). In Chapter 4 we study how to choose among several candidate SITs to obtain the most accurate cardinality estimates for input queries [BC02].

Second, we have to provide mechanisms to efficiently build SITs in a database system. While simple approaches might be sufficient in some cases, other situations require using customized algorithms. For instance, consider \( S_3 = \text{SIT}(o_{\text{totalprice}} \bowtie \text{orders} \bowtie \text{lineitem} \bowtie \text{part} \bowtie \text{supplier}) \). We can build \( S_3 \) by first executing its query expression \( \pi_{o_{\text{totalprice}}}(\text{orders} \bowtie \text{lineitem} \bowtie \text{part} \bowtie \text{supplier}) \) and then materializing \( S_3 \) over this intermediate result. Unfortunately, executing the query expression associated with \( S_3 \) can be expensive. In particular, since not all SITs might be already available in the database system, the optimizer could choose the very inefficient plan in Figure 3.4(a) for executing the generating query. For that reason, if efficiency is critical, we need alternative techniques that build approximate versions of SITs in a fraction of the original times while maintaining all the good properties of traditional SITs. In Chapter 5 we study several alternatives to accomplish this objective, which balance efficiency of construction and accuracy of the resulting SITs [BCG01a, BC03].

Third, we must address the problem of automatically identifying useful SITs to build and maintain. This procedure would free database administrators from the burden of
choosing which SITs to create and maintain. In general, this problem is difficult because we would like to recommend SITs without creating them in advance. Also, interactions between the set of available SITs make the analysis more complex. Finally, there is a tradeoff between the complexity of SITs and the set of queries that can be estimated with them. As an example, consider the following SITs:

\[ S_3 = \text{SIT}(\text{o\_totalprice}|\text{orders} \bowtie \text{lineitem} \bowtie \text{part} \bowtie \text{supplier}) \]
\[ S_4 = \text{SIT}(\text{o\_totalprice}|\text{orders} \bowtie \text{lineitem}) \]

While both \( S_3 \) and \( S_4 \) can be used to estimate the cardinality of the original query \( \sigma_{\text{o\_totalprice}>1,000,000} (\text{orders} \bowtie \text{lineitem} \bowtie \text{part} \bowtie \text{supplier}) \), SIT \( S_3 \) is likely to result in a better approximation than that of \( S_4 \). However, \( S_4 \) can be used to estimate the cardinality of additional queries that \( S_3 \) cannot, such as \( \sigma_{\text{o\_totalprice}>1,000,000} (\text{orders} \bowtie \text{lineitem}) \) and \( \sigma_{\text{o\_totalprice}>1,000,000} (\text{orders} \bowtie \text{lineitem} \bowtie \text{part}) \). Therefore, while complex SITs can be highly beneficial for a few specific queries, simpler SITs might be moderately useful for a larger class of queries. In Chapter 6 we study techniques to recommend SITs that strike a good balance between specificity and generality.

### 3.3 Conclusion

In this chapter we discussed the main simplifying assumptions used by optimizers during cardinality estimation, and showed how they can negatively affect the quality of the resulting query execution plans. We then introduced SITs, or statistics build on the result of execution query expressions, to diminish the propagation of errors through complex query plans. SITs can be used to directly model the distribution of tuples on intermediate nodes of query execution plans, therefore avoiding error-prone procedures that propagate base-table statistics. We then argued that the quality of execution plans can be significantly improved if the right SITs are present in the database system. Finally, we identified the three main challenges that need to be addressed for SITs to be effectively integrated with current optimizers. First, we need mechanisms to integrate SITs with
modern optimizers so that they can exploit this additional information during cardinality estimation. Second, we need to develop efficient techniques to materialize SITs in a database system. Third, we need to provide tools to help decide which SITs to build in a database system. In the following chapters we address each of these challenges in detail.
Chapter 4

Exploiting SITs

SITs are only useful if the optimizer is able to exploit them during query optimization. In this chapter we study techniques to enable SITs in a relational optimizer. We first introduce (Section 4.1) a general approach to exploit SITs that intercepts cardinality requests and rewrites the corresponding input plans to take advantage of the set of available SITs. Then, in Section 4.2 we focus on the (common) scenario of SPJ queries with aggregation and introduce a formal framework to reason with such SITs. Using this framework we can efficiently find the most accurate selectivity estimation for given SPJ input queries with optional \textit{group-by} clauses. Finally, in Section 4.3 we report experimental results on the efficiency and accuracy of the proposed techniques.

4.1 A Wrapper-based Architecture for Exploiting SITs

Generalizing the simple example of Section 3.2, we enable SITs in the optimizer by implementing a wrapper on top of the original cardinality estimation module of the RDBMS (see Figure 4.1). During the optimization of a single query, the wrapper is called many times, once for each different query sub-plan enumerated by the optimizer. Each time the query optimizer invokes the modified cardinality estimation module with a query sub-plan, we transform this input plan so that it exploits SITs. Then, we forward the
transformed query plan to the original cardinality estimation module. Finally, we obtain a potentially more accurate cardinality estimation for the original query, which is returned to the optimizer. The following is a list of properties that the proposed wrapper should satisfy:

- The transformed plan should exploit applicable SITs, so that its estimated cardinality is potentially more accurate than that of the original plan.
- The original cardinality estimation module should be able to take the transformed plan as input with only a few changes.
- The transformation should be efficient, since it is used for multiple sub-plans for each query that is optimized.

We now examine the main components of the proposed wrapper, which correspond to the square boxes in Figure 4.1. Specifically, Section 4.1.1 discusses the transformation step, while Section 4.1.2 identifies the portions of the original cardinality estimation module that need to be adjusted to process transformed query plans.

### 4.1.1 Transformation Step

We now describe the plan transformation step inside the wrapper of Figure 4.1. As we will see, we leverage work done in the context of materialized view matching to perform this step. For concreteness, we use the query in Figure 4.2(a) as a running example for most of this section.
4.1.1.1 Input Plan Analysis

Initially, we perform simple structure analysis on the input query. This analysis will later help identify which SITs to exploit. For instance, in Figure 4.2(b) we identify the tables and columns referenced in the query of Figure 4.2(a), together with the list of conjunctive predicates in the query. Next, we classify predicates as either filter or join predicates. We use the equality join predicates to generate column equivalence classes and also to get the set of table subsets that are joined (Figure 4.2(c)). The filter predicates are marked with an $F$ label, and the join predicates with a $J$ label. Tables $R$, $S$, and $V$ are joined using predicates $J_1$ and $J_3$, and tables $T$ and $U$ are joined using join predicate $J_2$. Similarly, columns $R:r$, $S:s$, and $V:v$ form one equivalence class, columns $T:t$ and $U:u$ form another equivalence class, and the remaining columns form singleton classes. A more complex analysis can be performed in this step, depending on the set of rewriting transformations that we apply later.

4.1.1.2 Application of Relevant SITs

For ease of notation, we represent the set of available SITs by using $SIT$-Sets, which basically group SITs by their generating query expressions. Specifically, the set of SITs \{SIT($a_1|Q$),...,SIT($a_n|Q$)\} can be compactly represented by a $SIT$-Set

$$Q \rightarrow S[a_1,\ldots,a_n]$$

where $S$ is a fresh identifier that “holds” each SIT($a_i|Q$). This notation reveals our intent to reuse ideas from the materialized view matching literature. In fact, a $SIT$-Set $Q \rightarrow S[a_1,\ldots,a_n]$ can be seen as a rewriting rule to replace an occurrence of $Q$ in the original query that exports attributes in \{$a_1,\ldots,a_n$\} with the identifier $S$.

During cardinality estimation for a query $q$, $S$ is treated as a base table having statistics on the \{$a_i$\} columns. Therefore, we check if $q$ can be rewritten using $S$ by employing a variety of well-known algorithms for materialized view matching (e.g., [CKPS95, GL01, PL00, Hal01]), and if so we generate a new equivalent query plan that incorporates SITs.
Incidentally, this is the procedure that we followed in Figure 3.3(b) to motivate the introduction of SITs.) Furthermore, note that such rewriting is exclusively used for cardinality estimation and not for plan generation. In other words, the transformed query plan is a temporary structure local only to the cardinality estimation module.
Exploiting Multiple SIT-Sets. In general, more than one SIT-Set may be applicable to a query expression. Consider Figure 4.2(d), which lists the SIT-Sets that are available in the example. For instance, the second SIT-Set has $R \bowtie_{r=v} V$ as its generating query and therefore can be applied to the query in Figure 4.2(a). Likewise, the third SIT-Set can be applied in conjunction with the second one (Figure 4.2(e)), resulting in the transformed query plan of Figure 4.2(f). In this case, the optimizer exploits SITs for all filter predicates except the one involving $S.b$, for which a traditional base-table histogram (if available) is used.

Ranking Alternative SIT-Sets. In the example above, using one SIT-Set does not interfere with the use of another. We now discuss a scenario in which the application of multiple SIT-Sets may not be mutually compatible. Suppose that SIT-Set $R \bowtie_{r=s} S \rightarrow RS[a, b]$ is added in Figure 4.2(d). Whenever SIT-Set $RST$ is applicable for a query, so is $RS$. However, $RST$ is favored over $RS$: while estimating the cardinality of the query transformed using $RST$ we make fewer independence assumptions compared to using $RS$. (As explained in Section 3.1.1, the independence assumption is the main source of errors during cardinality estimation.) Note, however, that $RS$ may be applicable in cases where $RST$ is not. These considerations have analogous counterparts in the literature of materialized view matching.

4.1.1.3 Incomplete SIT-Sets and Auxiliary SITs

A more complex scenario occurs when a given SIT-Set does not contain all SITs that are necessary from a view matching perspective. As we will see, we can still apply such SIT-Set at the expense of relying on some additional independence assumptions (but certainly on fewer than in the case that no SIT-Set is applied). Moreover, in some situations the missing attributes can be estimated with SITs from other “compatible” SIT-Sets. These issues are illustrated in the example below.

---

1 This SIT-Set represents unidimensional SITs over $a$ and $c$, and a multidimensional SIT over $(c, d)$. 

Example 7  Consider the following SIT-Sets:

\[
\begin{align*}
S \bowtie_{s=t} T & \quad \rightarrow \quad ST[b,c] \\
R \bowtie_{r=s} S \bowtie_{r=t} T & \quad \rightarrow \quad RST[a] \\
R \bowtie_{r=x} T & \quad \rightarrow \quad RT[a,c]
\end{align*}
\]

and suppose we want to estimate the cardinality of the query:

\[
\text{SELECT * FROM R, S, T WHERE } R.r=S.s \text{ AND } S.s=T.t \text{ AND } R.a<10 \text{ AND } T.c>20
\]

We can check that SIT-Set \( RST \) matches the input query (note that the join predicate \( R.r=S.s \) AND \( R.r=T.t \) in \( RST \)'s generating query is equivalent to the join predicate \( R.r=S.s \) AND \( S.s=T.t \) in the input query). We can then use SIT\((a|R \bowtie_{r=s} S \bowtie_{r=t} T)\) for the filter condition \( R.a < 10 \). In contrast, SIT\((c|R \bowtie_{r=s} S \bowtie_{r=t} T)\) is not available, so handling the condition \( T.c > 20 \) is more problematic. In fact, a traditional view matching algorithm would reject \( RST \) altogether because it omits some of the columns that are needed in the query. However, in our scenario we are not interested in executing the transformed query, but rather in using it to approximate the cardinality of the original query. Therefore, we can alternatively apply \( RST \), replacing \( R \bowtie_{r=x} T \) in the query with \( RST \), and use a base-table histogram for the missing attribute \( T.c \). By exploiting the \( RST \) SIT-Set as much as possible, we avoid relying on the independence assumption described earlier for column \( R.a \), and restrict the use of this assumption to just \( T.c \). Therefore, SIT-Set \( RST \) is an attractive alternative. We can potentially improve our size estimates further with SIT\((c|S \bowtie_{s=t} T)\) from SIT-Set \( ST \). With this SIT, we avoid assuming independence between \( T.c \) and \( T.t \) (we still assume independence between \( T.c \) and \( R \bowtie ST \)). We can use SIT\((c|S \bowtie_{s=t} T)\) because \( ST \) is compatible with \( RST \)'s generating query from a view matching perspective. In contrast, we cannot use SIT\((c|R \bowtie_{r=x} T)\) from SIT-Set \( RT \) instead because predicate \( R.r = T.x \) in its generating query does not match with predicate \( R.r = T.t \) in \( RST \). In general, exploiting additional SITs such as SIT\((c|S \bowtie_{s=t} T)\) positively impacts the accuracy of the resulting cardinality estimation.  

\[\square\]
The example above shows that applying strict materialized view rewriting to identify SITs during cardinality estimation is not appropriate in some cases, since such rewritings cannot account for the use of statistics such as $\text{SIT}(c|S \bowtie T)$. We refer to such additional SIT as an auxiliary SIT due to application of SIT-Set $RST$ to the input query.

In summary, to apply a SIT-Set $Q \rightarrow S[a_1, \ldots, a_n]$ to some query $q$, we first verify that $Q$ is compatible with $q$ and, if so, we determine a “rewriting” that uses the SIT-Set. Then, for each attribute in the tables of $Q$ that is (i) not covered by $S$ but (ii) still used in $q$ (e.g., attribute $T.c$ in the example above), we look for an auxiliary SIT that would provide the best alternative for estimation. Such SIT must come from a SIT-Set whose generating query is subsumed by the original SIT-Set’s generating query, or correctness would be compromised.

### 4.1.1.4 Putting it All Together

We now summarize the transformation step in the proposed wrapper of Figure 4.1. Conceptually, the goal is to find a rewriting that minimizes the number of independence assumptions used during cardinality estimation of the transformed query. Our attempt to minimize the number of applications of the independence assumption is justified since precisely independence assumptions are the main source of error for cardinality estimation. For efficiency purposes, we adopted a greedy heuristic to determine which SITs to exploit for an input query. At each iteration, we consider all the candidate SIT-Sets (and their corresponding auxiliary SITs), and select the alternative that results in the minimum number of independence assumptions. We repeat this procedure until no remaining SIT-Set matches the transformed query. In practice, we found that the greedy procedure obtains close to optimal transformations using a small amount of time. The procedure to transform an input plan is summarized in Figure 4.3.
4.1.2 Actual Estimation

After the original query plan is rewritten into an equivalent one as explained in the previous section, we forward the transformed plan to the original cardinality estimation module. We then obtain a (presumably better) cardinality estimation for the original query plan and we return that value to the optimizer. It is important to note once again that the transformed query is not used outside the cardinality estimation module. In fact, the execution engine would not be able to process transformed query plans since some of their referenced tables do not really exist in the database system.

To process transformed query plans, the original cardinality estimation module of a traditional optimizer needs some adjustments. First, we need to augment metadata information in the database system so that SIT-Sets are treated as hypothetical base tables, similar to how [CN98] treats hypothetical indexes. Also, we need to pass additional information to the original cardinality module to handle auxiliary SITs correctly. In particular, we need to augment the cardinality estimation module so that it accepts statistical hints that specify which SIT to use for some attributes in the query plan.
4.1.3 Conclusion

In this section we presented a general framework to exploit SITs during optimization. A wrapper that is placed on top of the original cardinality estimation module intercepts cardinality requests and transforms each input query plan into an equivalent one that exploits SITs as much as possible. The transformation step is based on a greedy procedure that selects which SITs to apply at each iteration, so that the resulting number of independence assumptions is minimized. Identifying whether or not a SIT is applicable to a given plan leverages well established results on materialized view matching, and therefore the proposed wrapper can be extended to complex queries.

Interestingly, the restricted family of SPJ queries with optional group-by clauses presents opportunities for further improving cardinality estimation, as illustrated next. Consider the query on the left of Figure 4.4, where \( R \triangleright S \) and \( R \triangleright T \) are (skewed) foreign-key joins, and each arrow in a join points to the table that holds the primary key. Suppose that only a few tuples in \( S \) and \( T \) satisfy predicates \( \sigma_{S,a<10}(S) \) and \( \sigma_{T,b>20}(T) \), yet most tuples in \( R \) join precisely with such tuples in \( S \) and \( T \). In the presence of the two SITs at the bottom of Figure 4.4, the wrapper introduced in this section explores the two maximal equivalent rewritings shown in the figure and eventually picks one of them as the transformed query plan. Each alternative exploits one available SIT and therefore takes into consideration correlations introduced by one of the skewed joins. Thus, the resulting estimations, although not perfect, have considerably better quality than when just using base-table statistics.

![Figure 4.4: Exploiting SITs using materialized view matching technology.](image-url)
Unfortunately, no transformation exploits both SITs simultaneously for cardinality estimation. The reason is that their query expressions are mutually exclusive from a view matching perspective. Moreover, this limitation is not an artifact of any specific procedure used to transform the query plan (e.g., the greedy approach of Section 4.1.1). Instead, it is a fundamental constraint that results from relying exclusively on materialized view matching to enumerate alternatives. In the example above, since both foreign-key joins are skewed towards the tuples that satisfy each filter condition, cardinality estimates obtained when using any one of the available SITs in isolation can still be an arbitrary underestimation (for that reason, even “averaging” alternative estimates does not work here). As we will see, the techniques introduced in the following section allow the optimizer to take advantage of both SITs simultaneously. Specifically, view matching is used as a tool to identify SIT's usage, but not as the main engine to enumerate alternatives.

4.2 A Formal Framework for Exploiting SPJ SITs

As explained before, the traditional concept of selectivity, used in conjunction with view matching technology, falls short of capturing the full set of opportunities to exploit SITs for some queries. In this section we focus on the class of SPJ queries with optional group-by clauses and present a comprehensive framework to efficiently identify the best alternative to estimate the cardinality of a given query for a set of available SITs. Below we summarize the contributions of this section:

Search space: As we will show in Section 4.2.1.1, there is a combinatorial explosion of alternatives to approximate cardinality values. Techniques that rely purely on materialized view matching to transform query plans can only explore relatively few possibilities, therefore missing some valuable opportunities to estimate cardinality values. In this section we introduce a formal framework to reason with selectivity values that allows us to easily identify and efficiently search the full space of alternatives to obtain accurate estimates.
**Efficiency:** During the optimization of a single query, many cardinality estimation requests refer to “similar” query plans. For instance, before requesting an estimation for the plan of Figure 4.4, the optimizer is likely to have already requested cardinality estimates for many of the plans’ components. The optimizer can then reuse previous computations during the optimization of the same query to satisfy new cardinality requests. The wrapper introduced in the previous section repeatedly transforms query sub-plans in isolation, and therefore does not take advantage of commonalities among sub-plans during optimization of a single query. As another contribution of this section, we present a dynamic programming algorithm, `getSelectivity`, that fully exploits such commonalities, and can be fairly easily integrated with a modern optimizer.

**Accuracy:** The metric used in the previous section to rank candidate transformations is the number of independence assumptions that are used during cardinality estimation. This metric, while simple and intuitive, is too coarse. Many different candidate transformations are given the same score, and usually ties are broken arbitrarily. As a final contribution of this section, we introduce `Diff`, a new metric to evaluate the expected benefit of exploiting particular SITs, which requires almost no overhead and results in close-to-optimal choices of SITs during cardinality estimation.

The key idea that we introduce is the concept of *conditional selectivity*, presented in Section 4.2.1. Conditional selectivity provides a formal framework to reason with selectivity values and makes it easy to identify and exploit SITs for cardinality estimation. We can draw an analogy between the use of SITs and work in traditional query optimization. During optimization, algebraic properties (e.g., associativity and commutativity of joins) provide different equivalent logical plans to represent the same input query. Similarly, conditional selectivity properties (Sections 4.2.1.1 and 4.2.1.2) result in many alternative ways (we call them *decompositions*) to estimate the same selectivity value. Moreover, optimizers choose among a set of access paths (e.g., sequential or index scans) to execute
each logical plan, and this choice is guided by a cost model. Section 4.2.2 introduces
an algorithm to identify the best alternative to approximate a selectivity value using
SITs. In particular, SITs can be seen as the implementation mechanism to approximate
decompositions (Section 4.2.2.3), and the search procedure (Section 4.2.2.2) is guided
by the estimated accuracy of each alternative decomposition (Section 4.2.2.4). In Sec-
tion 4.2.3 we show how the algorithm of Section 4.2.2 can be integrated with existing
query optimizers, and we discuss extensions to the basic framework in Section 4.2.4.

4.2.1 Conditional Selectivity

We now introduce the concept of conditional selectivity, which allows expressing a given
selectivity value in many different but equivalent ways. In this section we focus on
conjunctive SPJ queries, and in Section 4.2.4 we mention some extensions to handle more
general types of queries.

We represent an arbitrary SPJ query in a canonical form by first forming the cartesian
product of the tables referenced in the query, then applying all predicates (including
joins) to such cartesian product, and finally projecting out the desired attributes. Thus,
we represent a generic SPJ query as:

\[ q = \pi_{a_1, \ldots, a_n} (\sigma_{p_1 \land \ldots \land p_n} (R_1 \times \ldots \times R_n)) \]

where \( a_i \) are attributes of \( R_1 \times \ldots \times R_n \), and \( p_i \) are predicates over \( R_1 \times \ldots \times R_n \) (e.g.,
\( R_1.a \leq 25 \), or \( R_1.x = R_2.y \)).

Each set of predicates \( \{p_i\} \) that is applied to \( R_1 \times \ldots \times R_n \) results in the subset of tuples
that simultaneously satisfy all \( p_i \). We use bag semantics, so projections do not change
the size of the output, and therefore we omit them from consideration when estimating
 cardinalities. To estimate the size of the output, or its cardinality, we first approximate
the fraction of tuples in \( R_1 \times \ldots \times R_n \) that simultaneously satisfy all predicates \( p_i \) (i.e.,
the selectivity of all \( p_i \)), and then multiply such fraction by \( |R_1 \times \ldots \times R_n| \), which can
be obtained by simple lookups over the system catalogs. We now extend the traditional
definition of selectivity.
Definition 3 Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a set of tables, and $P = \{p_1, \ldots, p_j\}$ and $Q = \{q_1, \ldots, q_k\}$ be sets of predicates over $R_1 \times \ldots \times R_n$. The selectivity of $P$ with respect to $\sigma_{q_1 \land \ldots \land q_k}(R_1 \times \ldots \times R_n)$, denoted $Sel_{\mathcal{R}}(P|Q)$, is defined as the fraction of tuples in $\sigma_{q_1 \land \ldots \land q_k}(R_1 \times \ldots \times R_n)$ that simultaneously satisfy all predicates in $P$. Therefore,

$$Sel_{\mathcal{R}}(P|Q) = \frac{|\sigma_{p_1 \land \ldots \land p_j}(\sigma_{q_1 \land \ldots \land q_k}(R_1 \times \ldots \times R_n))|}{|\sigma_{q_1 \land \ldots \land q_k}(R_1 \times \ldots \times R_n)|}$$

If $Q = \emptyset$ we write $Sel_{\mathcal{R}}(P)$, which coincides with the original definition of selectivity.

We denote the set of tables referenced by a set of predicates $P$ as $\text{tables}(P)$, and the set of attributes mentioned in $P$ as $\text{attr}(P)$. To simplify the presentation, we also use “$P, Q$” to denote “$P \cup Q$” and “$p, Q$” to denote “$\{p\} \cup Q$”, where $p$ is a predicate and $P$ and $Q$ are sets of predicates. Finally, for a set of tables $\mathcal{R} = \{R_1, \ldots, R_n\}$, we use the notation $\mathcal{R}^x$ to refer to the cartesian product $R_1 \times \ldots \times R_n$.

Example 8 Consider query $q$:

$$\text{SELECT } * \text{ FROM } R, S, T$$
$$\text{WHERE } R.x = S.y \text{ AND } S.a < 10 \text{ AND } T.b > 5$$

The selectivity of $q$, $Sel_{\{R, S, T\}}(R.x = S.y, S.a < 10, T.b > 5)$, is the fraction of tuples in $R \times S \times T$ that satisfy all predicates. To obtain the cardinality of $q$, we multiply its selectivity by the factor $|R| \cdot |S| \cdot |T|$, which is obtained from system catalogs. Additionally, $\text{tables}(R.x = S.y, S.a < 10) = \{R, S\}$, and $\text{attr}(R.x = S.y, S.a < 10) = \{R.x, S.y, S.a\}$.

In general, for a given query $\sigma_{p_1 \land \ldots \land p_k}(R_1 \times \ldots \times R_n)$, our task is to estimate the value $Sel_{\{R_1, \ldots, R_n\}}(p_1, \ldots, p_k)$. Two properties of conditional selectivity values—presented below—allow us to express selectivity values in many equivalent ways and form the basis of the algorithm of Section 4.2.2 for obtaining the best selectivity estimation of a given query.
4.2.1.1 Atomic Decompositions

The key property of conditional selectivity values is atomic decomposition, which is based on the notion of conditional probability and unfolds a selectivity value as the product of two related selectivity values.

Property 1 (Atomic Decomposition) Given a set of tables $R$ and sets of predicates $P$ and $Q$:

$$Sel_R(P, Q) = Sel_R(P|Q) \cdot Sel_R(Q)$$

Proof: Using the definition of conditional selectivity, the above equality is expressed as:

$$\frac{|\sigma_{P \land Q}(R^x)|}{|R^x|} = \frac{|\sigma_P(\sigma_Q(R^x))|}{|\sigma_Q(R^x)|} \cdot \frac{|\sigma_Q(R^x)|}{|R^x|},$$

or equivalently,

$$|\sigma_{P \land Q}(R^x)| = |\sigma_P(\sigma_Q(R^x))|$$

which always holds in relational algebra. 

This property holds for arbitrary sets of predicates and tables, without relying on any assumption, such as independence. By repeatedly applying the atomic decomposition property, we can obtain a very large number of alternative rewritings for a given selectivity value, which we call decompositions. A decomposition of a given selectivity value $s$ is then an expression of the form $S_1 \cdot \ldots \cdot S_k$, where each $S_i = Sel_{R_i}(P_i|Q_i)$.

Lemma 1 The number of possible decompositions of $Sel_R(p_1, \ldots, p_n)$, denoted by $T(n)$, is bounded as follows for all $n \geq 1$:

$$0.5 \cdot (n + 1)! \leq T(n) \leq 1.5^n \cdot n!$$

Proof: The number of decompositions of $Sel_R(P)$, where $|P| = n$, is given by the following equation:

$$T(n) = \begin{cases} 1 & \text{if } n = 1 \\ \sum_{i=1}^{n} \binom{n}{i} \cdot T(n - i) & \text{otherwise} \end{cases}$$
In fact, for each \( 1 \leq i \leq n \), we can first decompose \( Sel_R(P) \) into \( Sel_R(P_1|P_2) \cdot Sel_R(P_2) \), where \( |P_1| = i \) and \( |P_2| = n - i \). Then, we recursively obtain all decompositions for \( Sel_R(P_2) \). After some manipulation, it follows that

\[
T(n+1) = \sum_{i=0}^{n} \binom{n+1}{i} \cdot T(i) = (n+1) \cdot T(n) + \sum_{i=0}^{n-1} \frac{n+1}{n+1-i} \cdot \binom{n}{i} \cdot T(i)
\]

The fraction in the summation is bounded as follows: \( 1 \leq \frac{n+1}{n+1-i} \leq \frac{n+1}{2} \) (for \( i=0 \) and \( i = n-1 \), respectively). From the first inequality, it follows that \( T(n+1) \geq (n+1) \cdot T(n) + T(n) = (n+2) \cdot T(n) \). By solving this simpler recurrence, we conclude that \( T(n+1) \geq 0.5 \cdot (n+2)! \), as desired. The remaining bound follows from an analogous analysis of the second inequality: \( T(n+1) \leq 1.5^{n+1} \cdot (n+1)! \).

If every possible factor \( Sel_R(P_i|Q_i) \) is calculated precisely, then every possible decomposition of \( Sel_R(P) \) evaluates to the same value (because we obtain each decomposition through a series of equalities). In real scenarios, however, we do not have available information to exactly calculate arbitrary \( Sel_R(P|Q) \) values. Instead, we maintain a set of SITs and use them to approximate decompositions of selectivity values. It follows that, depending on the available SITs, some decompositions might be more accurate than others. Suppose that we assign to each decomposition \( S \) of a selectivity value \( Sel_R(P) \) a measure of how accurately \( S \) can be approximated using the current set of available SITs. In that case, approximating \( Sel_R(P) \) can be seen as an optimization problem, in which we want to obtain the “most accurate” decomposition of \( Sel_R(P) \) for the given set of available SITs.

A naive approach to this problem would explore exhaustively all possible decompositions of \( Sel_R(P) \), estimate the accuracy of each decomposition and return the most accurate one. Unfortunately, this simple approach can be prohibitively expensive given the number of possibilities. We now introduce the notion of “separability,” a syntactic property of conditional selectivity values that can substantially reduce the space of decompositions without missing any useful one. Then, in Section 4.2.2 we present some natural restrictions on how the accuracy of a decomposition is measured, and then derive
a dynamic programming algorithm that efficiently returns the most accurate decomposition of a given selectivity value.

4.2.1.2 Separable Decompositions

The *separable decomposition* property can be seen as a syntactic notion of independence that allows simplifying a selectivity value whenever certain properties hold.

**Definition 4** We say that \( \text{Sel}_R(P|Q) \) is separable (with \( Q \) possibly empty) if we can find non-empty sets \( X_1 \) and \( X_2 \) such that \( P \cup Q = X_1 \cup X_2 \) and tables\((X_1) \cap \text{tables}(X_2) = \emptyset \). We also say that \( X_1 \) and \( X_2 \) separate \( \text{Sel}_R(P|Q) \).

**Example 9** Consider \( s = \text{Sel}_{\{R,S,T\}}(T.b=5, S.a < 10 | R.x=S.y) \). In this case, sets \( X_1 = \{T.b=5\} \) and \( X_2 = \{R.x=S.y, S.a < 10\} \) separate \( s \). In fact, \( \text{tables}(X_1) = \{T\} \) and \( \text{tables}(X_2) = \{R,S\} \). If we add predicate \( S.y = T.z \) to \( R.x=S.y \) in \( s \), the resulting expression \( \text{Sel}_{\{R,S,T\}}(T.b=5, S.a < 10 | R.x=S.y, S.y = T.z) \) is no longer separable. 

Intuitively, \( \text{Sel}_R(P|Q) \) is separable if \( \sigma_{P \land Q}(R^X) \) combines some tables by using cartesian products. It is important to note, however, that even if the original query does not use any cartesian product, after applying atomic decompositions some of its factors can become separable. For instance, consider \( s = \text{Sel}_{\{R,S\}}(R.a < 10, S.b > 5, R.x = S.y) \), which is not separable. After applying one atomic decomposition to \( s \) we obtain \( \text{Sel}_{\{R,S\}}(R.x = S.y | R.a < 10, S.b > 5) \cdot \text{Sel}_{\{R,S\}}(R.a < 10, S.b > 5) \), whose second factor is separable.

We now introduce the separable decomposition property.

**Property 2 (Separable Decomposition)** Suppose that \( \{P_1, P_2\} \) and \( \{Q_1, Q_2\} \) are partitions of \( P \) and \( Q \), respectively, and \( X_1 = P_1 \cup Q_1 \) and \( X_2 = P_2 \cup Q_2 \) separate \( \text{Sel}_R(P|Q) \). Let \( R_1 = \text{tables}(X_1) \) and \( R_2 = \text{tables}(X_2) \). Then,

\[
\text{Sel}_R(P|Q) = \text{Sel}_{R_1}(P_1|Q_1) \cdot \text{Sel}_{R_2}(P_2|Q_2)
\]
CHAPTER 4. EXPLOITING SITS

Proof: Let \( T = \mathcal{R} - (\mathcal{R}_1 \cup \mathcal{R}_2) \). We have that:

\[
\text{Sel}_\mathcal{R}(P|Q) = \text{Sel}_{\mathcal{R}_1 \cup \mathcal{R}_2 \cup T}(P_1, P_2|Q_1, Q_2) = \text{definition of Sel}
\]

\[
\frac{|\sigma_{P_1 \land P_2}(\sigma_{Q_1 \land Q_2}(\mathcal{R}_1^x \times \mathcal{R}_2^x \times T^x))|}{|\sigma_{Q_1 \land Q_2}(\mathcal{R}_1^x \times \mathcal{R}_2^x \times T^x)|} = \text{relational algebra}
\]

\[
\frac{|\sigma_{P_1 \land Q_1}(\mathcal{R}_1^x)| \cdot |\sigma_{P_2 \land Q_2}(\mathcal{R}_2^x)| \cdot |T^x|}{|\sigma_{Q_1}(\mathcal{R}_1^x)| \cdot |\sigma_{Q_2}(\mathcal{R}_2^x)| \cdot |T^x|} = \text{definition of Sel}
\]

\[
\text{Sel}_{\mathcal{R}_1}(P_1|Q_1) \cdot \text{Sel}_{\mathcal{R}_2}(P_2|Q_2)
\]

Example 9 (cont.) Since predicate sets \( \{T.b = 5\} \) and \( \{R.x = S.y, S.a < 10\} \) separate \( \text{Sel}_{\{R, S, T\}}(T.b = 5, S.a < 10 | R.x = S.y) \), we can rewrite such selectivity value as the product \( \text{Sel}_{\{R, S\}}(S.a < 10 | R.x = S.y) \cdot \text{Sel}_{\{T\}}(T.b > 5) \). In this case, both resulting factors are no longer separable. \[\]

Using the separable decomposition property, we make the following natural assumption concerning histograms.

Assumption 1 (Minimality for histograms) Suppose that \( \text{Sel}_\mathcal{R}(P|Q) \) is separable as \( \text{Sel}_{\mathcal{R}_1}(P_1|Q_1) \cdot \text{Sel}_{\mathcal{R}_2}(P_2|Q_2) \), and let \( H \) be a histogram that directly approximates \( \text{Sel}_\mathcal{R}(P|Q) \). We then assume that we can find histograms \( H_1 \) and \( H_2 \) approximating \( \text{Sel}_{\mathcal{R}_1}(P_1|Q_1) \) and \( \text{Sel}_{\mathcal{R}_2}(P_2|Q_2) \), respectively, such that: (i) \( H_1 \) and \( H_2 \) combined require at most as much space as \( H \) does, and (ii) the approximation of \( \text{Sel}_\mathcal{R}(P|Q) \) using \( H_1 \) and \( H_2 \) is as accurate as that from \( H \).

Consider, as a simple example, \( \text{Sel}_{\{R, S\}}(R.a < 10, S.b > 20) \), which is separable as \( \text{Sel}_{\{R\}}(R.a < 10) \cdot \text{Sel}_{\{S\}}(S.b > 20) \). In this situation, using two unidimensional histograms \( H(R.a) \) and \( H(S.b) \) to estimate each factor and then multiplying the resulting selectivity values assuming independence (which is correct in this case) is at least as accurate as directly using a two-dimensional histogram \( H(R.a, S.b) \) built on \( R \times S \). In fact,
the independence assumption holds in this case, so the joint distribution over \( \{R.a, S.b\} \) can be estimated correctly from unidimensional distributions over \( R.a \) and \( S.b \).

We then avoid maintaining SITs that directly approximate separable factors of decompositions, because such SITs can be replaced by more accurate and space-efficient ones. For that reason, we can safely discard from the search space all decompositions \( S = S_1 \cdot \ldots \cdot S_n \) for which some \( S_i \) is separable, without missing the most accurate decomposition. The separable decomposition property and the minimality assumption for histograms can substantially reduce the search space, since we avoid considering a large number of decompositions. However, in many cases the search space is still large. To make the optimization problem manageable, in the next section we impose certain restrictions on how the accuracy of a decomposition is measured, and derive an efficient procedure to obtain the most accurate decomposition for a given selectivity value.

### 4.2.2 Obtaining the Most Accurate Selectivity Estimation

As we saw in the previous section, the space of decompositions of a given selectivity value can be very large, even when ignoring decompositions with separable factors. We now derive a dynamic-programming algorithm that returns the most accurate decomposition for a selectivity value, provided that the function that measures the accuracy of decompositions is both **monotonic** and **algebraic** (see definitions below). As we will see, the complexity of the optimization problem is then drastically reduced with respect to the simple approach that enumerates all alternative decompositions.

#### 4.2.2.1 Ranking Candidate Decompositions

We now define the (estimated) **error** of a decomposition, which measures the accuracy of the available SITs approximating a given decomposition.

**Definition 5** Let \( s = \text{Sel}_R(p_1, \ldots, p_n) \) be a selectivity value, and \( S = S_1 \cdot \ldots \cdot S_k \) be a non-separable decomposition of \( s \), where \( S_i = \text{Sel}_{R_i}(P_i|Q_i) \). If we use SIT \( H_i \) to approximate \( S_i \), then \( \text{error}(H_i, S_i) \) is the level of accuracy of \( H_i \) approximating \( S_i \).
The value $\text{error}(H_i, S_i)$ is a positive real number, where smaller values represent better accuracy. The (estimated) overall error for $S = S_1 \cdots S_k$ is given by an aggregate function $E(e_1, \ldots, e_n)$, where $e_i = \text{error}(H_i, S_i)$. We restrict our attention to monotonic and algebraic [GCB+97] aggregate functions:

- A function $E$ is monotonic if every time that $e_i \leq e'_i$ for all $i$, we have that $E(e_1, \ldots, e_n) \leq E(e'_1, \ldots, e'_n)$. Monotonicity is a reasonable property for aggregate functions representing overall accuracy: if each individual error $e'_i$ is at least as high as error $e_i$, then we would expect the overall $E(e'_1, \ldots, e'_n)$ to be at least as high as $E(e_1, \ldots, e_n)$.

- A function $F$ is distributive if there is another function $G$ such that $F(x_1, \ldots, x_n) = G(F(x_1, \ldots, x_i), F(x_{i+1}, \ldots, x_n))$. Two examples of distributive aggregates are $\max$ (with $G=\max$) and $\text{count}$ (with $G=\text{sum}$). In general, $E$ is an algebraic function if there are distributive functions $F_1, \ldots, F_m$ and a function $H$ such that $E(x_1, \ldots, x_n) = H(F_1(x_1, \ldots, x_n), \ldots, F_m(x_1, \ldots, x_n))$. For instance, average is algebraic (with $F_1=\text{sum}$, $F_2=\text{count}$, and $H(x, y) = x/y$). To simplify the notation, we define $E_{\text{merge}}(E(x_1, \ldots, x_i), \ldots, E(x_{i+1}, \ldots, x_n)) = E(x_1, \ldots, x_n)$ for an algebraic function $E$. Therefore, $\text{avg}_{\text{merge}}(\text{avg}(1, 2), \text{avg}(3, 4)) = \text{avg}(1, 2, 3, 4)$.

Monotonicity imposes the principle of optimality for error values, and allows a dynamic programming strategy to find the optimal decomposition of $\text{Sel}_R(P)$. The principle of optimality states that the components of a globally optimal solution are themselves optimal. Therefore, we can find the most accurate decomposition of $\text{Sel}_R(P)$ by trying all atomic decompositions $\text{Sel}_R(P) = \text{Sel}_R(P'||Q) \cdot \text{Sel}_R(Q)$, recursively obtaining the optimal decomposition of $\text{Sel}_R(Q)$, and combining the partial results.

The key property of algebraic aggregates is that a small fixed-size vector can summarize sub-aggregations. We can then bound the amount of information to carry over between recursive calls to calculate error values. (Median, for instance, is not algebraic and requires space proportional to the input to exactly summarize a partial result.)
Note that we have not yet specified how to obtain the value $error(H, S)$ for SIT $H$ approximating a selectivity value $S$. In the next section we assume that we are given such a procedure and derive a general algorithm for getting the most accurate decomposition for a given selectivity value. Later, in Section 4.2.2.4, we introduce two alternative definitions for $error(H, S)$.

### 4.2.2.2 Algorithm getSelectivity

We now introduce $getSelectivity(R, P)$, a dynamic programming algorithm that computes the most accurate approximation of $Sel_R(P)$ for a given $error$ function. Algorithm $getSelectivity$ relies on the $error$ function being monotonic and algebraic, and avoids considering decompositions with separable factors. The pruning technique uses the fact that there is always a unique decomposition of $Sel_R(P)$ into non-separable factors of the form $Sel_{R_i}(P_i)$. In other words, if we start with $Sel_R(P)$ and repeatedly apply the separable decomposition property until no single resulting factor is separable, we always obtain the same non-separable decomposition of $Sel_R(P)$. We prove that lemma below.

**Lemma 2** There is a unique decomposition of $Sel_R(P) = Sel_{R_1}(P_1) \cdot \ldots \cdot Sel_{R_m}(P_m)$, called the standard decomposition of $Sel_R(P)$, such that each $Sel_{R_i}(P_i)$ is not separable.

**Proof:** Suppose that there are two standard decompositions of $S$, namely $S_1 = Sel_{R_1}(P_1) \cdot \ldots \cdot Sel_{R_m}(P_m)$ and $S_2 = Sel_{S_1}(Q_1) \cdot \ldots \cdot Sel_{S_n}(Q_n)$ of $Sel_R(P)$. Since $S_1$ and $S_2$ are different and $\cup_i P_i = \cup_j Q_j = P$, there must be a pair $(P_i, Q_j)$ such that $P_i \neq Q_j$ and $P_i \cap Q_j \neq \emptyset$. Let $X_{i,j} = P_i \cap Q_j$. Then, $X_{i,j}$ and $P_i - X_{i,j}$ separate $Sel_{R_i}(P_i)$. To prove that, let $T_{i,j} = tables(X_{i,j})$. It is fairly easy to show that $T_{i,j} \subseteq R_i$ and $T_{i,j} \subseteq S_j$. Also, by definition of $S_2$, $Q_j$ and $P - Q_j$ separate $Sel_R(P)$, so no single predicate from $P$ references attributes in $S_j$ and $R - S_j$ simultaneously. Therefore, no predicate in $P$ references attributes in $T_{i,j} \subseteq S_j$ and $(R_i - T_{i,j}) \subseteq (R - S_j)$, which essentially means that $X_{i,j}$ and $P_i - X_{i,j}$ separate $Sel_{R_i}(P_i)$. Analogously, $T_{i,j}$ and $Q_j - T_{i,j}$ separate $Sel_{S_j}(Q_j)$. Therefore, neither decomposition above is standard. \[\blacksquare\]
Algorithm \textit{getSelectivity} is shown in Figure 4.2.2.2. Lines 1-2 test whether the desired selectivity value was previously calculated, and if so returns it using a lookup in the memoization table. Otherwise, lines 4-7 handle the case in which $\text{Sel}_R(P)$ is separable. First, lines 4-5 obtain the (unique) standard decomposition of $\text{Sel}_R(P)$ and recursively call \textit{getSelectivity} for each factor $\text{Sel}_{R_i}(P_i)$. Then, lines 6-7 combine the partial results. Otherwise (if $\text{Sel}_R(P)$ is non-separable), lines 9-17 evaluate all atomic decompositions $\text{Sel}_R(P) = \text{Sel}_R(P'|Q) \cdot \text{Sel}_R(Q)$. For that purpose, line 11 recursively obtains the most accurate estimation (and the corresponding error) for $\text{Sel}_R(Q)$ and line 12 locally obtains the best SIT $H$ to approximate $\text{Sel}_R(P'|Q)$ among the set of available SITs (see Sections 4.2.2.3 and 4.2.2.4). If no SITs are available for approximating $\text{Sel}_R(P'|Q)$, we set $\text{error}_{p|Q} = \infty$ and continue with the next atomic decomposition. Lines 13-15 track the most accurate decomposition for $\text{Sel}_R(P)$, and after all atomic decompositions are explored, lines 16-17 obtain the most accurate estimation for $\text{Sel}_R(P)$. In all cases, before returning $\text{Sel}_R(P)$ and its associated error in line 19, \textit{getSelectivity} stores those values in the memoization table. As a byproduct of \textit{getSelectivity($R,P$)}, we get the most accurate selectivity estimation for every sub-query $\sigma_{P'}(R^x)$ with $P' \subseteq P$. We exploit these “free” selectivity estimates when integrating \textit{getSelectivity} with existing query optimizers in Section 4.2.3.

\textbf{Theorem 1} Algorithm \textit{getSelectivity($R,P$)} returns the most accurate approximation of $\text{Sel}_R(P)$ for a given definition of error among all non-separable decompositions.

\textbf{Proof: [Sketch]} We prove the theorem by induction on $|P|$. The base case ($|P| = 1$), is trivially verified. For the general case, if $S = \text{Sel}_R(P)$ is non-separable, line 10 exhaustively explores all atomic decompositions of $S$. Using the inductive hypothesis and the principle of optimality of error we prove the inductive step for this case. Otherwise, if $S$ is separable, we show that we do not miss any non-separable decomposition of $S$ by processing instead its standard decomposition (lines 4-7). Consider $S = \text{Sel}_R(P,Q)$, where $P$ and $Q$ separate $S$, and let $S' = \text{Sel}_R(P_1,Q_1|P_2,Q_2) \cdot \text{Sel}_R(P_2,Q_2)$ be an arbitrary atomic decomposition of $S$. Both factors of $S'$ are separable, so applying the separable decom-
We then know that due to memoization, only the first call for each subset of position property it follows that (the others are simple lookups). The running time of the complexity of $O(n)$.

Therefore, we do not miss any non-separable decomposition and again, by using the inductive hypothesis and the monotonicity of $error$, we prove the theorem.

The worst-case complexity of $getSelectivity(R, P)$, with $|P| = n$, is $O(3^n)$. In fact, the number of different calls of $getSelectivity$ is at most $2^n$, one for each subset of $P$. Due to memoization, only the first call for each subset of $P$ actually produces some work (the others are simple lookups). The running time of $getSelectivity$ for $k$ input predicates (not counting recursive calls) is $O(k^2)$ in lines 4-7 and $O(2^k)$ in lines 9-17. Therefore, the complexity of $getSelectivity$ is $O(\sum_{i=1}^{n} \binom{n}{i} \cdot 2^i)$, or $O(3^n)$. We contrast the worst-case complexity of $getSelectivity$, $O(3^n)$, with the lower bound of possible decompositions of

---

**getSelectivity** $(R:\text{ set of tables}, P:\text{ predicates over } R^*)$

**returns** $(\text{Sel}_R(P), error_P)$ such that $error_P$ is minimum.

01 if $(\text{Sel}_R(P)\text{ was already calculated})$
02 $(\text{Sel}_R(P), error_P) = \text{memoization_table_lookup}(P)$
03 else if $\text{Sel}_R(P)$ is separable
04 get the standard decomposition of $\text{Sel}_R(P) = \text{Sel}_R(P_1) \cdot \ldots \cdot \text{Sel}_R(P_n)$
05 $(S_P, error_P) = getSelectivity(R_i, P_i) \text{ for } 1 \leq i \leq n$
06 $S_P = S_{P_1} \cdot \ldots \cdot S_{P_n}$
07 $error_P = E_{merge}(error_{P_1}, \ldots, error_{P_n})$
08 else // $\text{Sel}_R(P)$ is non-separable
09 $error_P = \infty; \text{ bestH } = \text{ NULL}$
10 for each $P' \subseteq P, Q = P - P'$ // atomic decomposition $\text{Sel}_R(P'|Q) \cdot \text{Sel}_R(Q)$
11 $(S_Q, error_Q) = getSelectivity(R, Q)$
12 $(H, error_{P'|Q}) = \text{best SIT and error to approximate } \text{Sel}_R(P'|Q)$
13 // see Sections 4.2.2.3 and 4.2.2.4
14 if $(E_{merge}(error_{P'|Q}, error_Q) \leq error_P)$
15 $error_P = E_{merge}(error_{P'|Q}, error_Q)$
16 $bestH = H$
17 $S_{P'|Q} = \text{estimation of } \text{Sel}_R(P'|Q) \text{ using } bestH$
18 $S_P = S_{P'|Q} \cdot S_Q$
19 memoization_table_insert($P, S_P, error_P$)
20 return $(S_P, error_P)$

Figure 4.5: Obtaining the best approximation of a selectivity value.
Section 4.2.1.1, \( O((n+1)!)/3^n \) is \( \Omega(2^n) \), by using monotonic error functions we obtain an exponential decrease in the number of decompositions that are explored without missing the most accurate one. If many subsets of \( P \) are separable, the complexity of \textit{getSelectivity} is further reduced, since we solve strictly smaller problems independently. For instance, if \( \text{Sel}_R(P) = \text{Sel}_R_1(P_1) \cdot \text{Sel}_R_2(P_2) \), where \( |P_1|=k_1 \) and \( |P_2|=k_2 \), the worst case running time of \textit{getSelectivity} is \( O(3^{k_1} + 3^{k_2}) \), much smaller than \( O(3^{k_1+k_2}) \).

Line 12 in \textit{getSelectivity} obtains the SIT \( H \) that minimizes \( \text{error}(H, \text{Sel}_R(p|Q)) \) to approximate \( \text{Sel}_R(p|Q) \). The procedure consists of the following steps:

1. Obtain the set of candidate SITs that can approximate \( \text{Sel}_R(p|Q) \).
2. Select, from such set, the candidate \( \mathcal{H} \) that minimizes \( \text{error}(\mathcal{H}, \text{Sel}_R(p|Q)) \).

In the next sections we define the set of candidate SITs to estimate a selectivity value \( \text{Sel}_R(p|Q) \), and then we introduce two alternative definitions for error values.

### 4.2.2.3 Candidate SITs

In general, the statistical information needed to estimate a given \( \text{Sel}_R(P|Q) \) consists of multiple SITs: although simple filter conditions can be approximated using a single SIT, join predicates in general require at least two SITs. For simplicity, we slightly modify the notation to represent SITs as follows. Consider query expression \( q = \sigma_{p_1 \land \ldots \land p_k}(R^x) \). We alternatively use \( \text{SIT}_R(a_1, \ldots, a_j|p_1, \ldots, p_k) \) instead of \( \text{SIT}(a_1, \ldots, a_j|q) \). That is, we enumerate the set of predicates in \( q \) over \( R^x \), which coincides with the notation for selectivity values and simplifies the presentation. Therefore, \( H_R(a_1, \ldots, a_j|p_1, \ldots, p_k) \) is a histogram over attributes \( \{a_1, \ldots, a_j\} \) built on the result of executing \( \sigma_{p_1 \land \ldots \land p_k}(R^x) \). If there are no predicates \( p_i \) in \( q \) we write \( H_R(a_1, \ldots, a_j) \), which is a traditional base-table histogram. We now introduce the notion of \textit{predicate independence} that we use to define the set of candidate SITs to consider for approximating a given selectivity value.

\footnote{Specific error functions can take advantage of customized algorithms or data structures to improve efficiency.}
Definition 6 (Predicate Independence) Consider predicate sets $P_1$, $P_2$ and $Q$. We say that $P_1$ and $P_2$ are independent with respect to $Q$ if $Sel_R(P_1, P_2|Q) = Sel_{R_1}(P_1|Q) \cdot Sel_{R_2}(P_2|Q)$, where $R_1 = \text{tables}(P_1, Q)$ and $R_2 = \text{tables}(P_2, Q)$.

If $P_1$ and $P_2$ are independent with respect to $Q$, then $Sel_R(P_1|P_2, Q) = Sel_{R_1}(P_1|Q)$ holds as well. In fact, $Sel_R(P_1|P_2, Q) = \frac{Sel_R(P_1, P_2, Q)}{Sel_{R_2}(P_2|Q)} = \frac{Sel_{R_1}(P_1|Q) \cdot Sel_{R_2}(P_2|Q)}{Sel_{R_2}(P_2|Q)} = Sel_{R_1}(P_1|Q)$

We use independence between predicates as follows. Consider $Sel_R(p|Q)$ and suppose that there are no available SITs approximating $Sel_R(p|Q)$, but there are SITs approximating $Sel_R(p|Q')$, with $Q' \subset Q$. Then, we assume independence between $P$ and $Q'$ with respect to $Q-Q'$ and use those SITs to approximate $Sel_R(P|Q)$. Using this idea we now define the candidate set of SITs to approximate $Sel_R(P|Q)$.

Filter Predicates

Let $S=Sel_R(P|Q)$, where $P$ is a set of filter predicates, such as $\{R.a < 5 \land S.b > 8\}$. The candidate SITs to approximate $S$ are all $H_R(A|Q')$ that satisfy the following three properties:

1. $\text{attr}(P) \subseteq A$ (i.e., the SIT supports the predicates).

2. $Q' \subseteq Q$ (i.e., the SIT is consistent with the input query). In this case, we assume independence between $P$ and $Q-Q'$. In a traditional optimizer, $Q'=\emptyset$, so $P$ and $Q$ are always assumed to be independent.

3. $Q'$ is maximal, that is, there is no $H_R(A|Q'')$ available such that $Q' \subset Q'' \subseteq Q$. 
In principle, the set of candidate SITs can be defined in a more flexible way (e.g., including SITs of the form \( H_R(A|Q') \), where \( Q' \) subsumes \( Q \) [GL01]). Instead, we only consider the candidate SITs described above since they provide a good balance between the simplicity of the procedures to obtain candidate SITs and the quality of the resulting approximations (see Section 4.3 for experimental results).

**Example 10** Consider \( S = Sel_R(R.a < 5|p_1, p_2) \) and SITs \( H_R(R.a|p_1) \), \( H_R(R.a|p_2) \), \( H_R(R.a|p_1, p_2, p_3) \), and \( H_R(R.a) \). In this case, the set of candidate SITs for \( S \) include \( H_R(R.a|p_1) \) and \( H_R(R.a|p_2) \). In contrast, \( H_R(R.a) \) does not qualify since its query expression is not maximal. Similarly, \( H_R(R.a|p_1, p_2, p_3) \) does not qualify since it contains an additional predicate \( p_3 \).

**Filter and Join Predicates**

Consider a selectivity value \( Sel_R(P|Q) \) where \( P \) consists of both filter and join predicates (e.g., \( P = \{ T.a < 10, R.x = S.y, S.b > 5 \} \)). We use the following observation about histograms. Let \( H_1 = H_R(x,X|Q) \) and \( H_2 = H_R(y,Y|Q) \) be SITs. The join \( H_1 \bowtie_{x=y} H_2 \) returns not only the value \( Sel_R(x=y|Q) \) for the join, but also a new histogram \( H_3 = H_R(x,X,Y|x=y,Q) \) (see Section 2.2.2.2 for the procedure to obtain \( H_3 \)'s buckets). Therefore, we can use \( H_3 \) to estimate the remaining predicates involving attributes \( x(=y) \), \( X \), and \( Y \).

**Example 11** Consider \( Sel_R(R.a < 5, R.x = S.y|Q) \) and SITs \( H_1 = H_R_1(R.a|R.x, R.a|Q) \) and \( H_2 = H_R_2(S.y|Q) \). The join \( H_1 \bowtie_{R.x=S.y} H_2 \) returns the scalar selectivity value \( s_1 = Sel_R(R.x = S.y|Q) \) and also \( H_3 = H_R(R.a|R.x = S.y, Q) \). We then estimate \( s_2 \), the selectivity of filter predicate \( (R.a < 5) \) using \( H_3 \), and obtain \( Sel_R(R.x=S.y, R.a < 5|Q) = s_1 \cdot s_2 = Sel_R(R.a < 5|R.x = S.y, Q) \cdot Sel_R(R.x=S.y|Q) \) (implicitly applying an atomic decomposition).

As the example shows, we can approximate \( Sel_R(P|Q) \) by getting SITs covering all attributes in \( P \), joining such SITs, and finally estimating the remaining range predicates.
in \( P \). In general, the set of candidate SITs to approximate \( \text{Sel}_R(P|Q) \) is conceptually obtained as follows:

1. We transform all join predicates in \( P \) to pairs of wildcard selection predicates. For instance, predicate \( R.x = S.y \) in \( \text{Sel}_R(R.x = S.y, T.a < 10, S.b > 5|Q) \) is replaced by \( \{R.x = ?, S.y = ?\} \), and we obtain \( \text{Sel}_R(R.x = ?, S.y = ?, T.a < 10, S.b > 5|Q) \).

2. Let \( P' \) be the set of predicates obtained in the previous step. Because we replaced join predicates with filter predicates, the resulting selectivity value becomes separable. We then apply the separable decomposition property to obtain \( \text{Sel}_{R_1}(P'_1|Q_1) \cdot \ldots \cdot \text{Sel}_{R_k}(P'_k|Q_k) \), where no \( \text{Sel}_{R_i}(P'_i|Q_i) \) is separable.

3. Now, each \( \text{Sel}_{R_i}(P'_i|Q_i) \) contains only filter predicates in \( P'_i \), so we find each candidate set of SITs independently as described in Section 4.2.2.3.

To approximate the original selectivity value with the chosen SITs \( \{H_i\} \), we first join all \( H_i \) on the attributes mentioned in the wildcard predicates, and then estimate the actual range predicates over the result, as illustrated in Example 11.

Once we obtain the candidate set of SITs to approximate \( \text{Sel}_R(P|Q) \), we need to select the alternative that is expected to result in the most accurate estimation for \( \text{Sel}_R(P|Q) \). In other words, we have to choose SITs \( \mathcal{H} \) that are expected to minimize the value \( \text{error}(\mathcal{H}, \text{Sel}_R(P|Q)) \). In the next section we introduce two alternative formulations of \( \text{error} \) functions.

### 4.2.2.4 Error Functions

One of the critical subroutines of algorithm \( \text{getSelectivity} \) is \( \text{error}(\mathcal{H}, \mathcal{S}) \), which returns the estimated level of accuracy of approximating selectivity \( \mathcal{S} \) using SITs \( \mathcal{H} \). In fact, \( \text{error} \) guides the search for the decomposition with the minimal \( \text{error} \) value. We identify two requirements for any instance of an \( \text{error} \) function.
Efficiency: Evaluating $\text{error}(\mathcal{H}, \mathcal{S})$ values must be efficient, since $\text{error}$ is called repeatedly in the inner loop of $\text{getSelectivity}$. Very accurate but inefficient $\text{error}$ functions are not useful, since the overall optimization time would increase and therefore exploiting SITs would become less attractive. For instance, this requirement excludes a technique that looks at the actual data tuples to obtain exact $\text{error}$ values.

Coarseness of available information: At first sight, it is tempting to reformulate $\text{error}$ as a meta-estimation problem: to estimate the error between two data distributions (actual selectivity values and SIT-approximated selectivity values) we could maintain additional statistics, or meta-statistics, over the difference of such distributions. Therefore, estimating $\text{error}(\mathcal{H}, \mathcal{S})$ for a SIT $\mathcal{H}$ approximating a selectivity value $\mathcal{S}$ would be equivalent to approximate range queries over these meta-statistics. However, this approach is flawed, since if we do have such meta-statistics, we could combine them with the original SITs and obtain more accurate results in the first place. As an example, consider $\mathcal{H} = H_{\mathcal{R}}(R.a|p_1)$ approximating $\mathcal{S} = \text{Sel}_{\mathcal{R}}(R.a < 10|p_1,p_2)$. If we have available meta-statistics $\mathcal{M}$ to estimate values $\text{error}(\mathcal{H}, \text{Sel}_{\mathcal{R}}(c_1 \leq R.a \leq c_2|p_1,p_2))$, we can combine $\mathcal{H}$ and $\mathcal{M}$ to obtain new SITs that directly approximate $\text{Sel}_{\mathcal{R}}(R.a < 10|p_1,p_2)$.

Therefore, we need efficient and coarse mechanisms to estimate $\text{error}$ values. We can use existing information (e.g., system catalogs or characteristics of the input query) but not additional information created specifically for such purpose. We now revisit the metric used in Section 4.1 and adapt it to our framework, and then introduce an alternative formulation of $\text{error}$ that results in better estimations.

Counting Independence Assumptions: $\text{nInd}$

The $\text{nInd}$ error function is simple and intuitive, and is based on the observation that the independence assumption is the main source of errors during selectivity estimation. We define the overall $\text{error}$ of a decomposition $\mathcal{S} = \text{Sel}_{\mathcal{R}_1}(P_1|Q_1) \cdots \text{Sel}_{\mathcal{R}_n}(P_n|Q_n)$ when each factor is approximated, respectively, using $H_{\mathcal{R}_1}(A_1|Q'_1), \ldots, H_{\mathcal{R}_n}(A_n|Q'_n)$ ($Q'_i \subseteq Q_i$),
as the total number of predicate independence assumptions during the approximation, normalized by the maximum number of independence assumptions in the decomposition (to get a value between 0 and 1). In symbols,

$$n\text{Ind}({\{\text{Sel}_{R}(P_{i}|Q_{i}), H_{R}(A_{i}|Q_{i}'), i = 1, \ldots, n}\}} = \frac{\sum_{i} |P_{i}| \cdot |Q_{i} - Q_{i}'|}{\sum_{i} |P_{i}| \cdot |Q_{i}|}$$

Each term in the numerator represents the fact that $P_{i}$ and $Q_{i} - Q_{i}'$ are independent with respect to $Q_{i}$ (see Section 4.2.2.3), and therefore the number of predicate independent assumptions is $|P_{i}| \cdot |Q_{i} - Q_{i}'|$. In turn, each term in the denominator represents the maximum number of independent assumptions when $Q_{i}' = \emptyset$ (i.e., $|P_{i}| \cdot |Q_{i}|$). As a very simple example, consider $S = \text{Sel}_{R}(R.a < 10, R.b > 50)$ and decomposition $S = \text{Sel}_{R}(R.a < 10|R.b > 50) \cdot \text{Sel}_{R}(R.b > 50)$. If we use base table histograms $H(R.a)$ and $H(R.b)$, the error using $n\text{Ind}$ is $\frac{1 \cdot \left(1 - 0\right) + 1 \cdot \left(0 - 0\right)}{1 + 1} = 1/1 = 1$, or one out of one independence assumptions (between $R.a < 10$ and $R.b > 50$). $n\text{Ind}$ is clearly a syntactic definition and can be computed efficiently.

**Exploiting Data Distributions: Diff**

Although $n\text{Ind}$ is a simple metric, often many alternative SITs are given the same score, and $n\text{Ind}$ needs to break ties arbitrarily. This behavior is problematic when there are two or more available SITs to approximate a selectivity value, and while they all result in the same “syntactic” $n\text{Ind}$ score, the actual benefit of using each one of them is drastically different, as illustrated in the following example.

**Example 12** Consider $R \bowtie_{R.s = S.s} (\sigma_{S.a < 10} S) \bowtie_{S.t = T.t} T$, where both joins are defined between primary and foreign keys. Also consider the following factor of a decomposition for the given query: $S_{1} = \text{Sel}_{\{R,S,T\}}(S.a < 10 | R \bowtie S, S \bowtie T)$. Suppose that the only candidates to approximate $S_{1}$ are $H_{1} = H_{\{R,S\}}(S.a | R \bowtie S)$ and $H_{2} = H_{\{R,S\}}(S.a | S \bowtie T)$. If we use $n\text{Ind}$, $\text{error}(S_{1}, H_{1}) = \text{error}(S_{1}, H_{2}) = 1/2$, so in general each alternative is chosen at random 50% of the time. However, in this particular case, $H_{1}$ is always more helpful than $H_{2}$. In fact, since $S \bowtie_{S.t = T.t} T$ is a foreign-key join, the distribution of
attribute $S.a$ over the result of $S \bowtie_{S,T,T} T$ is exactly the same as the distribution of $S.a$ over base table $S$. Therefore, $S \bowtie_{S,T,T} S$ is actually independent of $S.a < 10$, and $H_2$ provides no benefit over the base histogram $H(S.a)$. ■

Inspired by the example above, we define $\text{Diff}$ as follows. Suppose we associate with each available SIT $H = H_R(R.a|Q)$ a single value $0 \leq \text{diff}_H \leq 1$. In particular, $\text{diff}_H = 0$ when the distribution of $R.a$ on the base table $R$ is exactly the same as that on the result of executing query expression $Q$. In contrast, $\text{diff}_H = 1$ when such distributions are very different (note that, in general, there are multiple possible distributions for which $\text{diff}_H = 1$, but only one for which $\text{diff}_H = 0$). Using $\text{diff}$ values, the $\text{Diff}$ error function generalizes $n\text{Ind}$ by providing a less syntactic notion of independence. In particular, the overall error value for a decomposition $S = \text{Sel}_{R_1}(P_1|Q_1) \cdot \ldots \cdot \text{Sel}_{R_n}(P_n|Q_n)$ when approximated using $H_1, \ldots, H_n$ is given by:

$$\text{Diff}((\text{Sel}_{R_i}(P_i|Q_i), H_i, i = 1, \ldots, n)) = \frac{\sum_i |P_i| \cdot |Q_i| \cdot (1 - \text{diff}_{H_i})}{\sum_i |P_i| \cdot |Q_i|}$$

Intuitively, value $|Q_i| \cdot (1 - \text{diff}_{H_i})$ in the numerator represents a “semantic” number of independent assumptions when approximating $S_i$ with $H_i$, and replaces the “syntactic” value $|Q_i - Q'_i|$ of $n\text{Ind}$. In fact, in Example 12, $\text{diff}_{H_1} = 0$, and $H_1$ effectively contributes the same as a base-table histogram $H(S.a)$, so in that case the error function is 1 (the maximum possible value). In contrast, for $H_2 = H_{[R,S]}(S.a|R \bowtie S)$, the more different the distributions of $S.a$ on $S$ and on the result of executing $R \bowtie S$, the more likely that $H_2$ encodes the dependencies between $S.a$ and $\{R \bowtie S, S \bowtie T\}$, and therefore the lower the overall error value.

We now define $\text{diff}_H$ for a given SIT $H$. For that purpose, consider $H = H_R(a|Q)$. If we denote $R' = \sigma_Q(R)$ (the result of evaluating $Q$ over $R$), we define $\text{diff}_H$ as

$$\text{diff}_H = 1/2 \cdot \sum_{x \in \text{dom}(a)} \left( \frac{f(R, x)}{|R|} - \frac{f(R', x)}{|R'|} \right)^2$$

where $f(R, x)$ is the frequency of value $x$ in $R$ ($\text{diff}_H$ is the squared deviation of frequencies between the base table distribution and the result of executing $H$’s query expression).
is fairly simple to show that $0 \leq \text{diff}_H \leq 1$, and that $\text{diff}_H$ satisfies the properties stated above. Values of $\text{diff}$ are calculated just once and are stored with each histogram, so there is no overhead at runtime. We can calculate $\text{diff}_{H\mathcal{R}(a\mid Q)}$ when we create $H\mathcal{R}(a\mid Q)$, but that might impose a certain overhead to the query processor to get values $f(R,a)$ if the tuples are not sorted by attribute $a$. Instead, we approximate $\text{diff}_H$ by carefully manipulating both $H$ and the corresponding base-table histogram (which, if it does not exists, can be efficiently obtained using sampling). The procedure is based on the technique of Section 2.2.2.2 to approximate the join of two histograms. We now describe how to obtain $\text{diff}_{H\mathcal{R}(a\mid Q)}$ using both histograms $H\mathcal{R}(a\mid Q)$ and base-table histogram $H\mathcal{R}(a)$:

1. **Align** buckets of $H\mathcal{R}(a)$ and $H\mathcal{R}(a\mid Q)$ (possibly splitting some buckets) so that each bucket boundary in the resulting $H\mathcal{R}(a)$ has a counterpart in the resulting $H\mathcal{R}(a\mid Q)$.

2. Consider in turn each pair of buckets $b_i$ (from $H\mathcal{R}(a)$) and $b_i'$ (from $H\mathcal{R}(a\mid Q)$). Let $f_i$ be the frequency of $b_i$, $dv_i$ the number of distinct values in $b_i$, and $\delta_i = f_i/dv_i$ the density of $b_i$. Similarly, let $f_i'$ be the frequency of $b_i'$, $dv_i'$ the number of distinct values in $b_i'$, and $\delta_i' = f_i'/dv_i'$ the density of $b_i'$. Using these values define:

$$C_i = \min(dv_i, dv_i') \cdot \left(\frac{\delta_i}{F} - \frac{\delta_i'}{F'}\right)^2$$

$$L_i = \max(dv_i - dv_i', 0) \cdot \left(\frac{\delta_i}{F}\right)^2$$

$$R_i = \max(dv_i' - dv_i, 0) \cdot \left(\frac{\delta_i'}{F'}\right)^2$$

where $F = \sum_i(f_i)$ and $F' = \sum_i(f_i')$. That is, for buckets $b_i$ and $b_i'$, the value $C_i$ estimates the terms $\left(\frac{\mathcal{I}(\mathcal{R}, x)}{|\mathcal{R}|} - \frac{\mathcal{I}(\mathcal{R}', x)}{|\mathcal{R}'|}\right)^2$ in the original definition of $\text{diff}$ for all values $x$ that are modelled in both buckets $b_i$ and $b_i'$ (specifically, $\min(dv_i, dv_i')$ of them). In turn, $L_i$ and $R_i$ estimate the same term for the remaining values in $b_i$ and $b_i'$ (if any) that have no matching tuples in the other bucket.

3. Finally, estimate $\text{diff}_{H\mathcal{R}(a\mid Q)}$ as follows:

$$\text{diff}_{H\mathcal{R}(a\mid Q)} = 1/2 \cdot \sum_i(L_i + C_i + R_i)$$
CHAPTER 4. EXPLOITING SITS

Of course, $Diff$ is just a heuristic ranking function and has some natural limitations. For instance, it uses a single number ($Diff$) to summarize the amount of divergence between distributions, and it does not take into account possible cancellation of errors among predicates. However, as we will see in Section 4.3, the additional information used by $Diff$ makes it much more robust and accurate than $nInd$ with almost no additional overhead.

4.2.2.5 A Final Example

To conclude this section, we revisit the example of Section 4.1.3 and show how our proposed technique results in better alternatives than those found by the techniques in Section 4.1, even when the same metric, $nInd$, is used by both strategies to select candidate SITs.

Example 13 Consider again the query of Figure 4.4. Using the notation of Section 4.2.1, we want to estimate the value $Sel_{(R,S,T)}(\sigma_a, \sigma_b, \bowtie_{RS}, \bowtie_{RT})$, where $\sigma_a$ and $\sigma_b$ represent the filter predicates over $S.a$ and $T.b$, respectively, and $\bowtie_{RS}$ and $\bowtie_{RT}$ represent the foreign-key join predicates. Using $nInd$, getSelectivity returns $S^* = Sel_{(R,S,T)}(\sigma_a | \bowtie_{RS}, \bowtie_{RT}) \cdot Sel_{(R,S,T)}(\bowtie_{RS} | \bowtie_{RT}) \cdot Sel_{(R,T)}(\bowtie_{RT})$ using, respectively, $H_{R,S}(S.a | \bowtie_{RS})$, $H_{R,T}(T.b | \bowtie_{RT})$, $\{H_R(R.s), H_S(S.s)\}$, and $\{H_R(R.t), H_T(T.t)\}$. Therefore, both available SITs are exploited simultaneously, producing a much more accurate cardinality estimate for the original query than any alternative produced by the wrapper proposed in Section 4.1. In fact, this is the best estimation we can obtain if those are the only available SITs. □

4.2.3 Integration with an Optimizer

In the previous section we showed how getSelectivity efficiently obtains the most accurate selectivity estimation for a given query. We now explain how getSelectivity can be integrated with existing rule-based optimizers.
For $q = \sigma_{p_1 \land \ldots \land p_k}(R^\times)$, \texttt{getSelectivity}(R, \{p_1, \ldots, p_k\}) returns the most accurate selectivity estimation for both $q$ and all its sub-queries (i.e., $Sel_R(P)$ for all $P \subseteq \{p_1, \ldots, p_k\}$). A simple approach to incorporate \texttt{getSelectivity} into an existing rule-based optimizer is to execute \texttt{getSelectivity before} optimization starts, and then use the resulting memoization table to answer selectivity requests over arbitrary sub-queries. This approach follows a pattern similar to that used by the Volcano optimization framework [GM93] to enumerate candidate sub-plans, in which a first step generates exhaustively all possible equivalent expressions and then, in a second phase, the actual search and pruning is performed. It was later established [Gra95] that this separation is not useful, since only a fraction of the candidate sub-plans generated during exploration is actually considered during optimization. Instead, newer approaches interleave an exploration by demand strategy with the search and pruning phase.

We follow a similar approach with \texttt{getSelectivity}. First, in Section 4.2.3.1 we describe Cascades, a framework used in current optimizers. Then, in Section 4.2.3.2 we show how we can couple \texttt{getSelectivity} with the search strategy of a Cascades-based optimizer.

### 4.2.3.1 Cascades-based Optimization

Cascades represented an important improvement over earlier extensible optimizers, and is the state-of-the-art rule-based optimization framework used in current optimizers such as Tandem’s NonStop SQL and Microsoft SQL Server.

During the optimization of an input query, a Cascades-based optimizer keeps track of many alternative sub-plans that could be used to evaluate the query. Sub-plans are grouped together into equivalence classes, and each equivalence class is stored as a separate node in a memoization table. Thus, each node in the memo contains a list of entries representing the logically equivalent alternatives explored so far. Each entry has the form

$$[op, \{input_1, \ldots, input_n\}, \{parameter_1, \ldots, parameter_k\}]$$

where $op$ is a logical operator, such as \texttt{join}, $input_i$ is a pointer to other memo node (another class of equivalent sub-queries), and $parameter_j$ is a parameter for $op$. 

Example 14 Consider the memo of Figure 4.6, that corresponds to an intermediate state of the optimization of:

```
SELECT * FROM R, S
WHERE R.x=S.y AND R.a<10 AND S.b>20
```

The node at the top of Figure 4.6 groups together all the already explored query plans that are equivalent to

\[(\sigma_{R.x=S.y}(\sigma_{S.b>20}(S))) R_1 / R_2\] (\sigma_{R.a<10}(R)).

The first entry in such node is

\[[\text{SELECT, } \{R_1 \bowtie_{R.x=S.y} (\sigma_{S.b>20}(S))\}, \{R.a < 10\}],\]

which corresponds to a filter operator, with parameter \(R.a < 10\), applied to the node that groups all equivalent expressions for sub-query \(R_1 \bowtie_{R.x=S.y} (\sigma_{S.b>20}(S))\). Analogously, the second entry corresponds to a join operator applied to two other nodes.

During optimization, each node in the memo is populated by applying transformation rules to the set of explored alternatives. Rules consist of antecedent and consequent patterns, and optional applicability conditions. The application of a given transformation rule is a complex procedure that involves:

1. Finding all bindings in the memo.
2. Evaluating the preconditions of the rule.
3. Firing the rule, where the pattern in the antecedent is replaced with that in the consequent.
4. Integrating the resulting expression (if it is new) into the memo table.

As a simple example, the first entry at the top node of Figure 4.6 could have been obtained from the second entry by applying the following transformation rule:

\[T_1 \bowtie (\sigma_P(T_2)) \Rightarrow \sigma_P([T_1] \bowtie [T_2]),\]

which pulls out selections above join predicates (\(T_1\) and \(T_2\) are placeholders for arbitrary sub-queries).
4.2.3.2 Integrating getSelectivity with Cascades

We now show how getSelectivity can be integrated with a Cascades-based optimizer. We first present an optimization to getSelectivity that does not compromise optimality but sometimes results in significant savings in execution time. Then, we introduce a heuristic pruning to couple getSelectivity with the optimizer’s exploration with little overhead.

Exploiting Restrictions in Available Statistics

If the optimizer restricts the set of available statistics (e.g., if it handles only unidimensional SITs), then getSelectivity can be implemented more efficiently without missing the most accurate decomposition. For unidimensional SITs, we can guarantee that no atomic decomposition $Sel_{\mathcal{R}}(P) = Sel_{\mathcal{R}}(P'|Q) \cdot Sel_{\mathcal{R}}(Q)$ with $|P'| > 1$ has a non-empty candi-
date set of SITs, and therefore is useful. In such a case, we can simply change line 10 in \(getSelectivity\) to:

\[
10 \quad \text{for each } P' \subseteq P, \text{ such that } |P'| \leq 1 \text{ do}
\]

without missing any decomposition. Using this optimization, the complexity of the algorithm \(getSelectivity\) is reduced from \(O(3^n)\) to \(O\left(\sum_{i} \binom{n}{i} \cdot i\right) = O(n \cdot 2^{n-1})\), and we are still guaranteed to return the most accurate selectivity estimations. As an interesting side note, this is the same reduction in complexity as obtained when we just enumerate linear join trees during optimization as opposed to bushy join trees.

**Coupling \(getSelectivity\) with the Optimizer’s Search Strategy**

We now explain how to further prune the search space of decompositions so that algorithm \(getSelectivity\) can be integrated with a Cascades-based optimizer. In particular, we couple the execution of \(getSelectivity\) with the optimizer’s own search strategy. As we will see, the suggested pruning is then guided by the optimizer’s own heuristics, and therefore might prevent \(getSelectivity\) from finding the most accurate estimation for some selectivity values. However, the overhead imposed to an existing optimizer is very small and, as we will see, the overall increase in quality can be substantial.

Consider an input SPJ query \(q = \sigma_{p_1 \land \ldots \land p_k}(R^x)\). As explained in Section 4.2.3.1, each node in the memoization table of a Cascades-based optimizer groups all alternative representations of a sub-query of \(q\). Therefore, we can estimate the selectivity of the sub-query represented by each node \(n\) in the memo, or \(Sel_R(P)\) for \(P \subseteq \{p_1, \ldots, p_k\}\). More importantly, each entry in \(n\) can be associated with a particular decomposition of the sub-query represented by \(n\). This issue is further illustrated below.

**Example 14 (cont.)** Consider again the node at the top of Figure 4.6, which groups all equivalent representations of \((\sigma_{R.a < 10}(R)) \bowtie_{R.x = S.y} (\sigma_{S.b > 20}(S))\). The second entry in such node (the join operator) can be associated with the following decomposition: \(Sel_{(R,S)}(R.x = S.y | R.a < 10, S.b > 20) \cdot Sel_{(R,S)}(R.a < 10, S.b > 20)\). Now, the first factor of this decomposition is approximated using available SITs as explained in Sections 4.2.2.3
and 4.2.2.4. The second factor is separable as $\text{Sel}_{\{R\}}(R.a < 10) \cdot \text{Sel}_{\{S\}}(S.b > 20)$. We obtain the estimated selectivity of each factor of the separable decomposition above by simply examining the corresponding memo nodes (the input of the join entry we are processing). Finally, we multiply such estimations and the first factor of the atomic decomposition $\text{Sel}_{\{R,S\}}(R.x=S.y|R.a < 10, S.b > 20)$ to obtain a new estimation for $\text{Sel}_{\{R,S\}}(R.x=S.y, R.a < 10, S.b > 20)$.

As hinted in the previous example, each entry $E$ in a memo node $n$ divides the set of predicates $P$ that are represented by $n$ into two groups: (i) the parameters of $E$, denoted $p_E$, and (ii) the predicates in the set input to $E$, denoted $Q_E = P - p_E$. We then associate the decomposition $\text{Sel}_R(P) = \text{Sel}_R(p_E|Q_E) \cdot \text{Sel}_R(Q_E)$ with entry $E$. It is fairly easy to show that for each operator in $E$, $\text{Sel}_R(Q_E)$ is separable into $\text{Sel}_{R_1}(Q^{1}_E) \cdot \ldots \cdot \text{Sel}_{R_k}(Q^{k}_E)$, where each $\text{Sel}_{R_i}(Q^{i}_E)$ is associated with the $i$-th input of $E$.

In summary, we restrict the set of decompositions in line 10 of $\text{getSelectivity}$ to exactly those induced by the optimization search strategy. Each time we apply a transformation rule that results in a new entry $E$ in the node associated with $\text{Sel}_R(P)$, we obtain the decomposition $S = \text{Sel}_R(p_E|Q_E) \cdot \text{Sel}_R(Q_E)$. If $S$ has the smallest error found so far for the current node, we update $\text{Sel}_R(P)$ using the new approximation. Therefore, the overhead imposed to a traditional rule-based optimizer is given by finding the most accurate SITs that approximate $\text{Sel}_R(p_E|Q_E)$ for each new entry $E$. We discuss the impact of such overhead experimentally in Section 4.3.

### 4.2.4 Extensions

We now briefly examine some extensions to the basic framework introduced in this section. In particular, we consider more general SPJ queries in Section 4.2.4.1, queries with group-by clauses in Section 4.2.4.2, and other extensions in Section 4.2.4.3.

---

3We know that those input nodes are already processed because of the search strategy followed by Cascades.
4.2.4.1 Non-Conjunctive SPJ Queries

In this section, we have so far focused on conjunctive SPJ queries. We now briefly discuss how to handle disjunctions (i.e., or conditions in the where clause). Using the identity 
\[ \sigma_{p_1 \lor p_2}(R^\times) = R^\times - (\sigma_{\neg p_1 \land \neg p_2}(R^\times)) \], we rewrite selectivity values as 
\[ Sel_R(p_1 \lor p_2) = 1 - Sel_R(\neg p_1, \neg p_2) \]. We then operate as usual, using the equality above whenever applicable. For instance, we rewrite 
\[ Sel_{\{R,S,T\}}(R.a < 5 \lor (S.b > 10 \land T.c=5)) \] as 
\[ 1 - Sel_{\{R,S,T\}}(R.a \geq 5, (S.b \leq 10 \lor T.c \neq 5)) \]. The second term can be separated as 
\[ Sel_{\{R\}}(R.a \geq 5) \cdot Sel_{\{S,T\}}(S.b \leq 10 \lor T.c \neq 5) \]. Now the second factor can be transformed again into 
\[ 1 - Sel_{\{S,T\}}(S.b > 10, T.c = 5) \], which is again separable. We continue until all factors are handled using the original approach. These transformations impose additional overhead to the optimizer, so in general it is important to carefully balance the complexity of the transformations and the benefit of using SITs.

4.2.4.2 SPJ Queries with Group-by Clauses

We now discuss SPJ queries with group-by clauses. In general, an SPJ query with a group-by clause can be specified as follows:

\[
\text{SELECT } b_1, \ldots, b_m \\
\text{FROM } R_1, \ldots, R_n \\
\text{WHERE } p_1 \text{ AND } \ldots \text{ AND } p_j \\
\text{GROUP BY } a_1, \ldots, a_k
\]

where each \( b_i \) is either included in \( \{a_1, \ldots, a_k\} \) or is an aggregate over some columns of \( R^\times \). The cardinality of \( q \) is equal to the number of groups in the output (i.e., the number of distinct values \( (a_1, \ldots, a_k) \) in \( \sigma_{p_1 \land \ldots \land p_j}(R^\times) \)), and can be obtained by multiplying \( |R^\times| \) by the selectivity of the query below:

\[
\text{SELECT DISTINCT } a_1, \ldots, a_k \\
\text{FROM } R_1, \ldots, R_n \\
\text{WHERE } p_1 \text{ AND } \ldots \text{ AND } p_j
\]
Thus, to approximate selectivity values of SPJ queries with group-by clauses, we need to estimate selectivity values of SPJ queries with set semantics (i.e., eliminating duplicate values). We next discuss some extensions to handle these queries.

### Conditional Selectivity

Let $P$ be a set of predicates, and $A$ be a set of attributes. We define $\text{tables}(P/A)$ as the set of tables referenced either in $A$ or in $P$, and $\text{attr}(P/A)$ as the attributes either in $A$ or referenced in $P$. We now extend the definition of conditional selectivity.

**Definition 7** Let $\mathcal{R} = \{R_1, \ldots, R_n\}$ be a set of tables, and $P$ and $Q$ be sets of predicates over $\mathcal{R}^\times = R_1 \times \ldots \times R_n$. Let $A$ and $B$ be sets of attributes over $\mathcal{R}$ such that $\text{attr}(P/A) \subseteq B$. Then, we define:

$$Sel_{\mathcal{R}}(P/A|Q/B) = \frac{|\pi_A^*(\sigma_P(\pi_B^*(\sigma_Q(R^\times))))|}{|\pi_B^*(\sigma_Q(R^\times))|}$$

where $\pi_A^*(R)$ is a version of the projection operator that eliminates duplicate values.

To simplify the notation of $S = Sel_{\mathcal{R}}(P/A|Q/B)$, we proceed as follows. If $B$ contains all attributes in $\mathcal{R}$, we omit $/B$ from $S$. Similarly, if $A = B$, we omit $/A$ from $S$. Finally, if $B$ contains all attributes in $\mathcal{R}$ and $Q = \emptyset$, we simply write $S = Sel_{\mathcal{R}}(P/A)$. The value $Sel_{\mathcal{R}}(P/A)$ is then the number of distinct $A$ values for tuples in $\sigma_P(\mathcal{R}^\times)$ divided by $|\mathcal{R}^\times|$. Therefore, for a generic SPJ query with a group-by clause such as the one at the beginning of this section, we are interested in estimating $Sel_{\mathcal{R}}(p_1, \ldots, p_j/a_1, \ldots, a_k)$.

### Decompositions

The atomic decomposition of Section 4.2.1.1 can be extended as follows:

**Property 3 (Atomic Decomposition)** Let $\mathcal{R}$ be a set of tables, $P$ be a set of predicates over $\mathcal{R}$, and $A$ be a set of attributes in $\mathcal{R}$. Then:

$$Sel_{\mathcal{R}}(P/A) = Sel_{\mathcal{R}}(P_1/A|P_2/B) \cdot Sel_{\mathcal{R}}(P_2/B)$$

where $P_1$ and $P_2$ partition $P$ and $\text{attr}(P_1/A) \subseteq B$. 
CHAPTER 4. EXPLOITING SITS

Proof: Replacing selectivity values by the fractions they represent, we get:

\[
\frac{\pi_A^*(\sigma_P(R^X))}{|R^X|} = \frac{\pi_A^*(\sigma_{P_1}(\pi_B^*(\sigma_{P_2}(R^X))))}{|R^X|}, \text{ or}
\]

\[
\pi_A^*(\sigma_P(R^X)) = \pi_A^*(\sigma_{P_1}(\pi_B^*(\sigma_{P_2}(R^X)))) \text{, or finally}
\]

which always holds in relational algebra, since \(P_1 \cup P_2 = P\) and \(\text{attr}(P_A) \subseteq B\) (so \(A \subseteq B\) and therefore \(A \cap B = A\)).

Analogous to the discussion in Section 4.2.3, we can integrate this generalized atomic decomposition with a rule-based optimizer that implements coalescing grouping transformations [CS94, CS96] for queries with group-by clauses. Coalescing grouping (see an example in Figure 4.7) is an example of push-down transformations, which typically allows the optimizer to perform early aggregation. In general, such transformations increase the space of alternative execution plans that are considered during optimization. The coalescing grouping transformation shown in Figure 4.7 is associated with the following instance of the atomic decomposition property: \(\text{Sel}_R(<A)/A = \text{Sel}_R(<A)/A \cdot \text{Sel}_R(<B)/B)\).

For the general case, we replace \(<B>\) in the equality above with the corresponding set of predicates.

![Figure 4.7: Simple Coalescing Grouping for group-by queries.](image)
CHAPTER 4. EXPLOITING SITS

For SPJ queries, we can use the atomic and separable decompositions alone to cover all transformations in a rule-based optimizer. In general, the situation is more complex for queries with *group-by* clauses. The separable decomposition property can be extended similarly as for the atomic property. Unfortunately, this is not always sufficient: some rule-based transformations require the operators to satisfy some semantic properties. For instance, the invariant grouping transformation [CS94] shown in Figure 4.8 requires, among others restrictions, that the join predicate be defined over a foreign key of $R_1$ and the primary key of $R_2$. In this case, we need to introduce specific decompositions that mimic such transformations. For instance, consider again Figure 4.8. Using atomic decomposition we obtain that $Sel_R(\bowtie /A) = Sel_R(\bowtie |/A) \cdot Sel_R(/A)$. However, if we can apply the invariant group transformation, $Sel_R(\bowtie |/A) = Sel_R(\bowtie)$ also holds. For that reason, we can use the decomposition $Sel_R(\bowtie /A) = Sel_R(\bowtie) \cdot Sel_R(/A)$, which can be easily integrated with a rule-based optimizer. This transformation is not valid for arbitrary values of $P$ and $A$, but instead holds whenever the invariant grouping transformation can be applied.

![Figure 4.8: Invariant Grouping for *group-by* queries.](image)

**Candidate SITs**

In the context of SITs as histograms, we can still exploit traditional histogram techniques provided that they record not only the frequency but also the number of distinct values per bucket. Consider again the atomic decomposition for the transformation in Figure 4.7, i.e., $Sel_R(\bowtie /A) = Sel_R(/A |\bowtie /B) \cdot Sel_R(\bowtie /B)$. We can use $H_R(A |\bowtie /B)$ to approximate $Sel_R(/A |\bowtie /B)$. In general, to approximate $Sel_R(/A B/Q_B)$ we use
some candidate SITs of the form $H_R(A|Q'/B)$ where $Q' \subseteq Q$. These considerations are similar to those of Section 4.2.2.3. In this scenario, however, we need to consider more special cases and take into account particular rule transformations and assumptions that vary among specific systems.

### 4.2.4.3 Other Extensions

We conclude this section by briefly commenting on further extensions for SITs that can be used to handle queries with complex filter conditions as well as queries with *having* clauses. Consider the following query:

```sql
SELECT * FROM orders
WHERE ship-date - place-date < 5
```

which asks for orders that were shipped less than 5 days after they were placed. We cannot obtain a good cardinality estimation of the query above by just using unidimensional base-table histograms over columns `ship-date` and `place-date`. The reason is that single-column histograms fail to model the strong correlation that exists between the ship and place dates of any particular order. A multidimensional histogram over both `ship-date` and `place-date` might help in this case, but only marginally. In fact, most of the tuples in the two-dimensional space `ship-date × place-date` will be very close to the diagonal `ship-date = place-date` because most orders are usually shipped a few days after they are placed. Therefore, most of the tuples in `orders` will be clustered in very small sub-regions of the rectangular histogram buckets. The uniformity assumption inside buckets would then be largely inaccurate and result in estimations that are much smaller than the actual cardinality values.

Alternatively, in this section we propose to extend the scope of SIT($A|Q$) to obtain better cardinality estimates for queries with complex filter expressions. Specifically, we allow $A$ to be a column expression over $Q$. A column expression over $Q$ is a function that takes as input other columns accessible in the SELECT clause of $Q$ and returns a scalar value. For instance, a SIT that can be used to accurately estimate the cardinality of the
query above is \( H = \text{SIT}(\text{diff-date}|Q) \) where the generating query \( Q \) is defined as:

\[
\text{SELECT ship-date - place-date as diff-date} \\
\text{FROM orders}
\]

In fact, each bucket in \( H \) with range \([x_L \ldots x_R]\) and frequency \( f \) specifies that \( f \) orders were shipped between \( x_L \) and \( x_R \) days after they were placed. Thus, the cardinality of the query above can be estimated accurately with a range query \([-\infty \ldots 5]\) over \( H \).

This simple idea can also be used to specify SITs that help estimating the cardinality of queries with \textit{group-by} and \textit{having} clauses. Consider the query:

\[
\text{SELECT A, sum(C)} \\
\text{FROM R} \\
\text{GROUP BY A} \\
\text{HAVING avg(B)<10}
\]

which conceptually groups all tuples in \( R \) by their \( A \) values, then estimates the average value of \( B \) in each group, and finally reports only the groups with an average value smaller than 10. The cardinality of this query can be estimated using \( H_2 = \text{SIT}(\text{avgB} \mid Q_2) \), where the generating query \( Q_2 \) is defined as:

\[
\text{SELECT avg(B) as avgB} \\
\text{FROM R} \\
\text{GROUP BY A}
\]

In this case, \( H_2 \) is a histogram in which each bucket with range \([x_L \ldots x_R]\) and frequency \( f \) specifies that \( f \) groups of tuples from \( R \) (grouped by \( A \) values) have an average value of \( B \) between \( x_L \) and \( x_R \). Therefore, the cardinality of the original query above can be estimated with a range query, with range \([-\infty \ldots 10]\), over \( H_2 \).

We note that the generalizations introduced in this section provide further opportunities to exploit SITs during optimization. Unfortunately, the extensions discussed above additionally complicate the different algorithms and quality metrics in the conditional selectivity framework discussed in previous sections. We do not explore these extensions further in the rest of this chapter.
4.3 Experimental Evaluation

In this section, we experimentally study getSelectivity using both $nInd$ and $Diff$ as the underlying error functions, and compare it against the wrapper-based approach of Section 4.1. As we will see, using conditional selectivity values results in more accurate estimations than when solely relying on materialized view matching to exploit SITs. Also, when using $Diff$, the getSelectivity approximations are close to being optimal. For our experiments we use the following setting:

**Data Sets:** We generate a synthetic database with a snowflake schema, as shown in Figure 4.9. Each node in the figure represents a 1,000- to 1,000,000-tuple table with 4 to 8 attributes, and each edge represents a foreign-key join. Attribute values are generated with different degrees of skew and correlation. To increase the complexity of the data sets, some foreign-key joins do not verify referential integrity. In such situations, for a foreign-key join $R \bowtie S$, we chose a certain percentage of tuples in $R$ (between 5% and 20%) and replace the join attribute in those tuples with NULL values. The choice of the dangling tuples is either random or correlated with attribute values.

**Workloads:** Each workload consists of 100 randomly generated SPJ queries, with parameters $J$ (number of join predicates) and $F$ (number of filter predicates). In our experiments, we set $F$ to three\(^4\) and vary $J$ from 3 to 7. For each query, we choose filter predicates such that the selectivity of each one is between 0.04 and 0.06. If the

\(^4\)We also conducted experiments with more filter predicates and obtained similar results.
query result is empty, we progressively stretch the filter ranges until at least one tuple is present in the result. We also used predicates with selectivities in the range $[0.4 - 0.6]$ and obtained similar results, but we believe that those queries are less frequent in real applications, so we omit those results.

**Available SITs:** We experiment with different pools of available SITs. Each SIT is a unidimensional MaxDiff histogram [PIHS96] with at most 200 buckets. Each set of SITs $J_i$ contains all histograms of the form $H_R(a|Q)$, where $Q$ consists of at most $i$ join predicates, and both $Q$ and $a$ are syntactically present in some query in the workload ($J_0$ contains all and only base-table histograms). The number of available SITs ranged from 82 (for $J_1$) to 680 (for $J_7$).

**Techniques Compared:** We implemented the technique described in Section 4.1, which we refer to as $GVM$ (for Greedy View Matching), and several variations of the algorithm $getSelectivity$ of Section 4.2. In particular, we implemented $nInd$, the variation that counts the number of independence assumptions, and $Diff$, the variation that uses additional information about the data distribution. We also implemented $Opt$, which uses the actual difference between the true selectivity $S$ and the approximation using $\mathcal{H}$ to define $error(\mathcal{H}, S)$: this definition of $error$ is the best possible one, but is only of theoretical interest since it cannot be implemented efficiently (it inspects the actual data). Finally, we implemented $noSit$, which is a baseline technique that mimics current optimizers and exploits base-table histograms only.

**Metrics:** We compare the accuracy of the different techniques as follows. For each query $q$ in the workload, we first estimate the cardinality of each sub-query of $q$ using the different techniques. Then, we evaluate each sub-query to obtain its actual cardinality value, and finally obtain the average estimation error over all sub-series of $q$. We repeat this procedure for each query $q$ in the workload and average the individual results.
4.3.1 Comparison between the Wrapper Approach and $getSelectivity$

We compare $GVM$ against $nInd$, so that the error function in $getSelectivity$ agrees with that of the greedy procedure in $GVM$. Any difference in accuracy is then due to $getSelectivity$ exploring the full search space, and not caused by an improved error function such as $Diff$. Figure 4.10 shows a two-dimensional graph, where each point represents one query in a workload consisting of three- to seven-way join queries. The $x$- and $y$-axes represent the absolute cardinality error for each query using $GVM$ and $getSelectivity$, respectively. As shown in the figure, all points lie under the line $x=y$, which means that $nInd$ results in consistently better cardinality estimates than $GVM$. The reason is that the search space in $GVM$ is a strict subset of the space of decompositions explored by $nInd$. In addition, $GVM$ uses a greedy technique to iteratively select SITs, which further reduces the set of decompositions explored. Although $nInd$ is based on the same metric to rank candidate SITs, it can result in absolute errors that are as much as 80% lower than those of the $GVM$ technique.

![Figure 4.10: Accuracy of $nInd$ vs. $GVM$.](image)

To compare the efficiency of both techniques, we proceed as follows. Both $GVM$ and $getSelectivity$ share the same view matching algorithm in their inner loops to select SITs ($GVM$ during the greedy procedure, and $getSelectivity$ as a subroutine in line 12). To
avoid comparing execution times of different pieces of code with various levels of optimization, we chose to compare the average number of calls to the view matching routine\(^5\), shown in Figure 4.11 for both GVM and getSelectivity and for different workloads. The dynamic programming algorithm used by getSelectivity results in drastically fewer calls, despite searching the whole space of decompositions. Although GVM is more efficient than getSelectivity for an isolated sub-plan, it does not exploit commonalities between different sub-plans, and therefore results in as much as 5 times as many calls to the view matching routine as getSelectivity does.

![Figure 4.11: Efficiency of nInd vs. GVM.](image)

In summary, we experimentally established the superiority of getSelectivity (using nInd) over GVM, both in terms of accuracy and efficiency, for SPJ queries. We next take a closer look at getSelectivity and compare the error functions nInd and Diff.

### 4.3.2 Accuracy of getSelectivity

Figure 4.12 shows the average absolute error for workloads of three-, five-, and seven-way join queries. In all cases, the absolute error is reduced drastically when all SITs with join query expressions \([J_7]\) are available (e.g., the average absolute error is reduced

\(^5\)We also compared pure execution times in our specific implementation and the trends were similar to those presented here.
from 62,466 to 1,679 in Figure 4.12(a)). In particular, \( \text{Diff} \) is very close to the optimal strategy \( \text{Opt} \), and results in considerably more accurate estimations than \( \text{nInd} \). For \( \text{Diff} \), the largest reduction in error occurs for \( J_1 \) and \( J_2 \). Thus, in our experiments, SITs with two- and three-way join query expressions are responsible for most of the accuracy gains.

### 4.3.3 Efficiency of \textit{getSelectivity}

Figure 4.13 shows the average execution time of \textit{getSelectivity} for different input workloads and with \( \text{Diff} \) as the error function. (The results for \textit{getSelectivity} using \( \text{nInd} \) are very similar.) We partition the execution time of \textit{getSelectivity} into two components. The decomposition analysis is the time spent to process the different decompositions and choose the best candidate SITs. The histogram manipulation corresponds to line 16 of \textit{getSelectivity} and measures the actual estimation of selectivity values using the selected SITs. The reason for this division is that different techniques choose different histograms to estimate the same selectivity values. Different histograms take different amounts of time to estimate the same predicates, so we report these components separately.

Figure 4.13 shows that, in general, \textit{getSelectivity} results in a small overhead over \textit{noSit}, under 6 milliseconds in all scenarios (under 4 milliseconds if we also consider the histogram manipulation component). We note that the overhead of \textit{getSelectivity} is proportional to the number of candidates considered in the algorithm, and scales gracefully with the number of available SITs. As an example, in Figure 4.13, \textit{getSelectivity} executes in around 6 milliseconds for \( J_1 \) (with 82 SITs), and in around 9 milliseconds for \( J_7 \) (with 680 SITs).

In conclusion, we expect that the overhead of \textit{getSelectivity} in a real optimizer will be very small, since Figure 4.13 accounts for just a portion of the total optimization time. Other components, such as a sophisticated rule-based exploration engine or the cost estimation module that uses cardinality estimates as input also contribute to the overall optimization time.
Figure 4.12: Absolute error for $getSelectivity$ with different input workloads.
(a) three-way join queries.

(b) five-way join queries.

(c) seven-way join queries.

Figure 4.13: Execution times for getSelectivity with different input workloads.
4.3.4 Exploiting Multidimensional Statistics using *getSelectivity*

So far, we have only used unidimensional histograms in our experiments. Therefore, the number of recursive calls in *getSelectivity* was moderate, since we only explored decompositions of the form $\text{Sel}_R(P|Q) \cdot \text{Sel}_R(Q)$ for $|P| = 1$. When multidimensional histograms are available, the *getSelectivity* search space includes many new decompositions, which increases the estimation overhead. We now quantify this overhead for decompositions that require at most $d$-dimensional histograms, for a parameter $d$. Specifically, we simulate the existence of multidimensional histograms and evaluate the overhead introduced by the additional decompositions that are explored. Figure 4.14(a) shows the execution time of *getSelectivity* when we only count the decomposition analysis. Figures 4.14(b) and 4.14(c) show, respectively, the number of nodes explored in the memo tables, and the number of decompositions evaluated when we vary the histogram dimensionality. We can see that execution time and number of decompositions (and to a lower extent, the number of nodes explored) are highly correlated. When we augment the dimensionality of the available histograms, the execution time for *getSelectivity* increases as well. This is especially true for 7-way join queries, for which the average execution time of *getSelectivity* increases to over 160 ms. Figure 4.14(c) also shows the important pruning in the number of decompositions when we just consider unidimensional histograms (*getSelectivity* explores just 713 decompositions without missing the optimal one, against over 11,000 for 10-dimensional histograms).

The above discussion represents a worst-case scenario, and many optimizations are possible to reduce the number of decompositions to be explored. In high dimensions, the majority of decompositions do not have any matching SITs (e.g., *getSelectivity* explores over 11,000 decompositions for 10-dimensional histograms, but the number of available histograms would be orders of magnitude smaller). We can slightly modify *getSelectivity* so that the enumeration of alternative non-separable decompositions is guided by the set of available histograms, without performing an exhaustive search. In this case, the efficiency of *getSelectivity* would be comparable to the case shown in Figure 4.13.
CHAPTER 4. EXPLOITING SITS

Figure 4.14: Expected overhead when using multidimensional histograms.

(a) Execution time (only decomposition analysis).

(b) Number of nodes explored.

(c) Number of decompositions explored.
4.4 Conclusion

In this chapter we showed how to exploit SITs during cardinality estimation to improve the execution-plan quality. The main contributions of this chapter are:

- We showed how to integrate SITs into state-of-the-art query optimizers. Specifically, we “wrap” the optimizer’s cardinality estimation module so that it can intercept cardinality requests and transform the input query plans into equivalent ones that exploit SITs. The transformation step leverages results on materialized view matching, so the wrapper can be easily extended to incorporate new results in view matching to further improve SIT’s usage.

- For the specific case of SPJ queries with optional group-by clauses:
  - We introduced a formal framework to reason about selectivity values, using the notion of conditional selectivity. With this framework, we can easily identify and search the full space of decompositions that approximate selectivity values for given SITs. By using conditional selectivity values, the optimizer can consider many critical decompositions that are otherwise missed by techniques that rely exclusively on materialized view matching technology.
  - We designed a dynamic programming algorithm, \textit{getSelectivity}, that efficiently returns the most accurate selectivity estimation for an input query. Algorithm \textit{getSelectivity} can be integrated with existing optimizers by coupling its execution with the optimizer’s own search strategy. Our experiments show that \textit{getSelectivity} results in accurate estimations, while its expected overhead is small.
  - We introduced a new metric, \textit{Diff}, to approximate the accuracy of alternative decompositions. \textit{Diff} takes into consideration the actual data distribution and therefore results in better estimations that “syntactic-only” metrics such as \textit{nInd}. In effect, the decompositions that \textit{getSelectivity} produces using \textit{Diff} are
very close in accuracy to those from hypothetical strategies that have access to perfect information.

To conclude this chapter, we note that the wrapper-based technique discussed in Section 4.1 is more general in scope than the techniques based on conditional selectivity of Section 4.2. In fact, the former can handle more complex query expressions (e.g., recursive or nested queries) provided that the necessary view matching infrastructure is in place. In contrast, the latter is specifically tailored to the family of SPJ queries with optional \textit{group-by} clauses, and needs to be carefully extended to deal with more complex queries (e.g., see the necessary modifications in Section 4.2.4.2). We believe that a combination of both approaches is ideal in practical situations, where the optimizer can exploit conditional selectivity values whenever possible, and otherwise uses the default approach based on materialized view matching.
Chapter 5

Building SITs

The previous chapter introduced techniques to exploit SITs in an RDBMS during cardinality estimation. As discussed, SITs generally lead to much more accurate estimates than those derived using base-table statistics only. Our discussion has so far assumed an existing set of available SITs. We now turn to the fundamental issue of actually materializing SITs. (Then, Chapter 6 will discuss how to choose what SITs to build.) A natural strategy to create a specific SIT is as follows. We first evaluate the SIT’s generating query and materialize the resulting set of tuples. Then, we derive the required SITs over this temporary result. Finally, after statistics have been created, the intermediate table is discarded. SITs can then be updated periodically just as base-table statistics do. This simple procedure for building SITs can handle arbitrary query expressions and is independent of the particular structure of SITs (e.g., histograms). Furthermore, this procedure can be the basis for extending the `CREATE STATISTICS` statement in SQL: instead of specifying the table name of the query, a more general query expression can be specified.

Unfortunately, the simple approach described above might not be desirable in all situations. In fact, the execution of the generating queries associated with SITs can be expensive. This situation is further aggravated because, in general, multiple SITs might not be already materialized in the system, which in turn might result in poor execution
CHAPTER 5. BUILDING SITS

plans for the generating queries of the SITs to be built. In this chapter we explore alternative strategies to create SITs. Although our new techniques are not as general as the simple approach described above in terms of supported generating queries or internal structure of the resulting SITs, our new procedures can create SITs much more efficiently than by fully evaluating the generating query expressions. The rest of this chapter is structured as follows. First Section 5.1 presents a feedback-driven architecture to build SITs with little overhead as queries are answered by the RDBMS. We show how to progressively refine SITs over arbitrary query expressions provided that such expressions (or similar ones) are periodically executed by the RDBMS. As a result, SITs are created in parallel with actual query executions, with small overhead. Our discussion will focus on STHoles histograms as the structure of SITs: STHoles histograms rely on a workload-based algorithm to refine buckets that can be adapted with almost no changes to our proposed architecture. The ideas of Section 5.1, though, can also be used with other statistics that can be refined using query feedback (e.g., the histograms of [AC99]).

Next, Section 5.2 introduces a complementary family of techniques to create SITs. These techniques work off-line and balance efficiency of construction and accuracy of the resulting SITs. Specifically, we focus on SITs with acyclic-join generating queries, and then show how to extend the basic approach to more complex generating queries. The proposed techniques create a SIT(A|Q) by effectively working on a (close) approximation of Q, which is obtained without executing Q. We claim (and verify experimentally) that if we use a SIT built over a close approximation of its query expression, the resulting cardinality estimates are close to those obtained when SITs are created over the exact (and more expensive to obtain) intermediate distribution.

5.1 A Feedback-Driven Approach to Build and Refine SITs

In this section we present a low-overhead strategy for building and refining SITs online in an RDBMS. Our technique intercepts intermediate results while queries are being executed, and gathers useful aggregated information that is later used to refine histogram
CHAPTER 5. BUILDING SITS

buckets. This approach does not need to access the underlying data distribution directly, hence avoiding materializing query expressions associated with SITs. Instead, SITs are incrementally refined as a “side effect” of query execution.

Figure 5.1 shows our proposed architecture for building and refining SITs. For each incoming query, the optimizer exploits available SITs during cardinality estimation (as described in Chapter 4) and produces a query execution plan. This plan is passed to the execution engine, where it is evaluated. During the execution of a query plan, streams of tuples— which correspond to intermediate results—are passed from one operator to another in the execution plan (or from the root operator in the plan back to the user). We then collect some aggregated information on intermediate results in the query execution plan, and use such information to tune relevant SIT buckets. Specifically, we augment the execution plan with run-time counters that summarize intermediate results. The refined SITs are then used during optimization of subsequent queries, and the cycle repeats.

Figure 5.1: Feedback-driven architecture for SIT refinement.
A technical issue that needs to be addressed in this framework is how to reconcile logical and physical plans in the database system. In fact, the aggregated information is gathered for nodes in the physical plan that is being executed. To refine SITs, we need to track the physical plan back to the logical plan that originated it. This problem is simple in some cases (e.g., tracking a specific variation of hash join to the corresponding algebraic join operator), and more complex in other situations (e.g., for index-scans with additional filter predicates). Some practical considerations to address the mismatch between logical and physical plans can be found in [SLMK01]. We now provide additional details on the proposed feedback-based architecture for building and refining SITs.

Gathering Statistics. An important decision in feedback-based architectures is to decide where to collect information from executed query plans. If the SITs to materialize are fixed in advance (e.g., as the result of the automatic techniques for choosing SITs of Chapter 6) the answer is relatively simple. We check each candidate SIT($A|Q$), and we gather information on plan node $q$ if (i) $q$ matches $Q$, and (ii) $q$ contains an additional filter condition over $A$ (as an example, Figure 5.2 shows an execution plan and three SITs that can be refined using feedback from intermediate tuple streams). In contrast, if the SITs to build and maintain are not predetermined, we can choose the nodes in the execution plans for which we will collect information using the following simple criteria:

- Select nodes with sub-plans involving at most $k$ operators (i.e., favor simple SITs).
- Select nodes with the largest estimated cost (i.e., favor SITs over expensive plans).
- Select all nodes in the query plan.

Types of Statistics Gathered. Gathering information from query feedback has an associated run-time overhead. The magnitude of this overhead depends on the kind of statistics that are maintained. For instance, just recording query-result cardinality has low overhead [BCG01b]. However, counting the number of distinct values in a tuple stream is usually expensive and requires large amounts of memory (even if approximations
are tolerated). To keep overhead low, we gather only query result cardinalities and omit statistics on distinct result values. Interestingly, sometimes the cardinality of the result of some query might be equivalent to the number of distinct values in the result of simpler sub-queries. For instance, the number of tuples in the result of a query with a “GROUP BY $a_1, \ldots, a_n$” clause is equal to the number of distinct values in the projection, over attributes $\{a_1, \ldots, a_n\}$, of the sub-plan below the GROUP BY operator.

We now focus on $STHoles$ as the histogram of choice to refine SITs. $STHoles$ histograms are designed to rely on query feedback and can be easily adapted to the architecture of Figure 5.1. (As mentioned earlier, the proposed architecture can be used with other statistics as well.) We define $STHoles$ histograms in Section 5.1.1. Then, in Section 5.1.2 we present algorithms to refine $STHoles$ histograms that exploit query feedback. Finally, in Section 5.1.3 we report an experimental evaluation of $STHoles$ histograms.

### 5.1.1 STHoles Histograms

As mentioned in Section 2.2.1.2, $STHoles$ histograms are built by using feedback from the execution of user queries, but without looking at the actual data distributions. This characteristic of the refinement procedure is useful in the context of base-table histograms [BCG01a]. Interestingly, this feature can also be exploited to build and refine SITs: since feedback is gathered from intermediate nodes in the query execution plan, we can build SITs without materializing their associated query expressions.
STHoles histograms introduce a new partitioning scheme that allows buckets to overlap, therefore better capturing complex data regions with uniform tuple density than more rigid partitioning schemes. Each bucket in an STHoles histogram identifies a rectangular range in the data domain, similar to other histogram techniques. However, unlike traditional histograms, STHoles histograms identify bucket sub-regions with different tuple density and “pulls” them out from the corresponding buckets. Hence a bucket $b$ can have holes, which are themselves first-class histogram buckets. These holes are $b$’s children, and their bounding boxes are disjoint and completely enclosed in $b$’s bounding box.

An STHoles histogram can then be conceptually seen as a tree structure, where each node represents a bucket. The volume of bucket $b$ is defined as $v(b) = vBox(b) - \sum_{b' \in \text{children}(b)} vBox(b')$, where $vBox(b)$ is the volume of $b$’s bounding box. Given a histogram $H$ and a range query $q$, the estimated cardinality of $q$, denoted $est(H, q)$, is given by the following expression:

$$est(H, q) = \sum_{b_i \in H} f_i \cdot \frac{v(q \cap b_i)}{v(b_i)}$$

where $f_i$ is the frequency of bucket $b_i$ and $v(q \cap b_i)$ denotes the volume of the intersection of $q$ and $b_i$ (which is in general different from $vBox(b_i)$).

**Example 15** Figure 5.3 shows an STHoles histogram with four buckets. The root of the tree is bucket $b_1$, with frequency 100. It has two children, namely buckets $b_2$ and $b_3$, with frequencies 500 and 1,000, respectively. Finally, bucket $b_3$ has one child, $b_4$, with frequency 200. The region associated with a particular bucket excludes that of its descendants, which can be thought of as “holes” in the parent space. Note that the region captured by bucket $b_1$ (shaded in Figure 5.3) is not rectangular, even though we only use rectangular buckets for partitioning the space. A query that covers the lower half of bucket $b_3$ is estimated to return nearly 1,000 tuples, even when it covers half of $b_3$’s bounding box, because the other half is not considered as part of $b_3$. More precisely, there is another bucket ($b_4$) that covers that region.
5.1.2 STHoles Construction Algorithm

An interesting property of STHoles histograms is that they can be built and refined without actually examining the underlying data, but rather by just inspecting query results. To build and refine an STHoles SIT(A|Q), we start with the best initial approximation of SIT(A|Q) available in the system. For instance, if a base-table histogram H(A) is available, we use it as a starting point. Alternatively, we start with an empty histogram that contains no buckets.

Let $q$ be a node in the query execution plan that we use to refine an STHoles SIT(A|Q). We define $R(q)$ as the region in the SIT’s domain that might be refined using feedback from $q$. Specifically, $R(q)$ denotes the portion of $A$’s domain (specified as a range predicate) that is “covered” by the filter predicate in $q$. For instance, consider SIT($S.a, T.b | R \bowtie S \bowtie T \bowtie U$) in Figure 5.2. The corresponding execution plan in the figure matches the SIT’s generating query ($R \bowtie S \bowtie T \bowtie U$) and additionally has filter predicates $S.a < 10$ and $T.b > 5$. In this case, $R(q) = \{(S.a, T.b) | -\infty < S.a < 10 \land 5 < T.b < \infty\}$. Every tuple in the result of $q$, projected over attributes $S.a$ and $T.b$, belongs to $R(q)$.

To refine an STHoles histogram $H$, we first intercept the result stream of tuples from the corresponding node in the query execution plan and count how many tuples lie inside each bucket of $H$. Next we determine which regions in the data domain can benefit from using this new information, and refine the histogram by “drilling holes,” or zooming into the buckets that cover the region identified by the query plan. Finally, we consolidate
the resulting histogram by merging similar buckets so that we do not exceed our fixed 
storage budget. We summarize the steps for refining an \textit{STHoles} histogram below, and 
next we give more details on each component.

\begin{tabular}{|l|}
\hline
\textbf{BuildAndRefine} (H: \textit{STHoles} histogram, q: query plan node) \\
1 Count, for each bucket \( b_i \) in H, the number of tuples in \( R(q) \cap b_i \) \\
2 Identify candidate holes in H \hspace{5mm} // Section 5.1.2.1 \\
3 Drill candidate holes as new buckets in H \hspace{5mm} // Section 5.1.2.2 \\
4 Merge superfluous buckets in H \hspace{5mm} // Section 5.1.2.3 \\
\hline
\end{tabular}

\subsection{5.1.2.1 Identifying Candidate Holes}

In general, \( R(q) \) intersects some buckets only partially (\( R(q) \) is defined by the filter 
predicates over the attributes of the \textit{STHoles} histogram). For each such bucket \( b_i \), we 
know the exact number of tuples in \( R(q) \cap b_i \) by inspecting the results for \( q \). Intuitively, if 
\( R(q) \cap b_i \) has a \textit{disproportionately} large or small fraction of the tuples in \( b_i \), then \( R(q) \cap b_i \) 
is a candidate to become a hole of bucket \( b_i \). Hence, each partial intersection of \( R(q) \) and 
a histogram bucket can be used to improve the quality of the corresponding histogram.

\textbf{Example 16} Figure 5.4 shows a bucket \( b \) with frequency \( f_b = 100 \). Suppose that \( T_b = 90 \) 
tuples from the result stream of node \( q \) lie in \( R(q) \cap b \). We then deduce that bucket \( b \) is 
significantly skewed, since 90\% of its 100 tuples are located in a small fraction of its 
volume. We can improve the accuracy of the histogram if we create a new bucket \( b_n \) 
by “drilling” a hole in \( b \) that corresponds to the region \( R(q) \cap b \) and adjust \( b \) and \( b_n \)'s 
frequencies accordingly, as illustrated in Figure 5.4. \hfill \| 

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure5_4}
\caption{Drilling a hole in bucket \( b \) to improve the histogram quality.}
\end{figure}
If the intersection of range $R(q)$ and bucket $b$ is rectangular (as in Figure 5.4) we consider $R(q) \cap b$ as a candidate hole and proceed as in the previous example. However, in general it is not always possible to create a hole in a bucket $b$ to form a new bucket $R(q) \cap b$. The problem is that some children of $b$ might be taking some of $b$’s space, and therefore the bounding box of $R(q) \cap b$ might not be rectangular anymore (e.g., the intersection between $R(q)$ and $b_p$ has an $L$ shape in Figure 5.4, due to bucket $b$). To avoid this problem, we could simply discard such cases from consideration while refining $STHoles$. This approach, however, typically reduces the accuracy of the resulting histograms, because the intersection $R(q) \cap b$ is not rectangular for a large percentage of cases. Another alternative is to decompose the non-rectangular region $R(q) \cap b$ into a covering set of smaller candidate holes with rectangular shape. While this option is conceptually attractive, it is usually expensive and can generate a large number of new candidate buckets for each input query. For those reasons, we chose a middle ground to approximate the shape of $R(q) \cap b$ when it is not rectangular. Essentially we shrink $R(q) \cap b$ to a large rectangular sub-region that does not partially intersect with the bounding box of any other bucket. We then estimate the number of tuples in this sub-region assuming uniformity. That is, if $T_b$ is the number of tuples in $R(q) \cap b$ and $c$ is the result of shrinking $R(q) \cap b$, we estimate $T_c$, the number of tuples in $c$, as $T_c = T_b \cdot \frac{v(c)}{v(R(q) \cap b)}$.

**Example 17** Figure 5.5 shows a four-bucket histogram and the progressive shrinking of the initial candidate hole $c = R(q) \cap b$. At the beginning, the buckets that partially intersect with $c$, called participants in the algorithm below, are $b_1$ and $b_2$ ($b_3$ is completely included in $c$). We first shrink $c$ along the “vertical” dimension so that the resulting candidate hole $c'$ does not intersect with $b_1$ anymore. Then, we shrink $c'$ along the “horizontal” dimension so that the resulting candidate hole $c''$ does not intersect with $b_2$. At this point there is no bucket that partially intersects with $c''$. The resulting candidate hole $c''$ is rectangular and covers a significant portion of the original $R(q) \cap b$ region.
More generally, the procedure for shrinking the intersection of bucket $b$ and range $R(q)$ is shown in Figure 5.6. We adopted a greedy approach to shrink a non rectangular region $R(q) \cap b$ in which, at each iteration, we contract the current candidate hole $c$ along the dimension that results in the smallest reduction in area for $c$. In general, we identify the candidate new holes to refine a given histogram by invoking $\text{shrink}(b_i, q, T_{b_i})$ for all buckets $b_i$ such that $R(q) \cap b_i \neq \emptyset$, where $T_{b_i}$ is the number of tuples in the result stream from $q$ that lie inside bucket $b_i$.

![Figure 5.5: Shrinking a candidate hole $c = R(q) \cap b$.](image)

<table>
<thead>
<tr>
<th>Shrink (b: bucket, q: query plan node, T_b: number of tuples in bucket b)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 $c = R(q) \cap b$</td>
</tr>
<tr>
<td>2 participants = ${b_i \in \text{children}(b): c \cap b_i \neq \emptyset \land b_i \not\subseteq c}$</td>
</tr>
<tr>
<td>3 while (participants $\neq \emptyset$)</td>
</tr>
<tr>
<td>4 Select bucket $b_i \in$ participants and dimension $j$ such that shrinking $c$ along $j$ by excluding $b_i$ results in the smallest reduction of $c$</td>
</tr>
<tr>
<td>5 Contract $c$ along dimension $j$</td>
</tr>
<tr>
<td>6 Update participants</td>
</tr>
<tr>
<td>7 $T_c = T_b \ast v(c) / v(R(q) \cap b)$ // adjust frequency</td>
</tr>
<tr>
<td>8 Return candidate hole $c$ with frequency $T_c$</td>
</tr>
</tbody>
</table>

Figure 5.6: Shrinking the intersection of a bucket and a range.

### 5.1.2.2 Drilling Candidate Holes as New Histogram Buckets

In the previous section we identify candidate new holes to refine an $STHoles$ histogram. Each candidate hole $c$ with frequency $T_c$ is completely included in some bucket $b_i$ and does not intersect partially with any child of $b_i$. We now show how to effectively “drill”
such candidates as new histogram buckets. For this purpose, we identify three possible scenarios:

1. Candidate hole $c$ and bucket $b_i$ reference the same region in the data domain (i.e., their bounding boxes are the same). In this case, we simply replace $b_i$’s frequency with $c$’s updated information, $T_c$.

2. Candidate hole $c$ covers all $b_i$’s remaining space. In this (rare) situation, $c$ and $b_i$’s bounding boxes are not the same, but $c$ covers all of $b_i$’s space that is not already taken by $b_i$’s children (see Figure 5.7 for an example of this scenario). Adding a new child of $b_i$ would result in wasted space, since one histogram bucket would effectively cover an empty region. To avoid this situation, we first eliminate bucket $b_i$ from the histogram (transferring $b_i$’s children to $b_i$’s parent) and then attempt again to drill $c$.

3. Candidate hole $c$ covers some of $b_i$’s remaining space. In this default scenario, we create a new child of $b_i$, denoted $b_n$, with a bounding box equal to that of $c$, and $f_{b_n} = T_c$. We then migrate all of $b_i$’s children whose bounding boxes are completely included in $c$ so they become children of the new bucket $b_n$. Finally, we adjust the frequency of $b_i$ to restore, whenever possible, the previous frequency counts. That is, if we had enough tuples in $b_i$ (i.e., $f_{b_i} \geq T_c$), we subtract $T_c$ from $f_{b_i}$. Otherwise, we simply set $f_{b_i}$ to zero.

Figure 5.8 shows the procedure to drill a new candidate hole in an $STHoles$ histogram.
5.1.2.3 Merging Buckets

The previous section showed how we can refine an STHoles histogram by drilling holes into existing buckets. In doing so, we might temporarily exceed the maximum number of histogram buckets. Hence, after adding buckets, we need to consolidate the resulting histogram by reducing the number of buckets. We do so by merging similar ones, so that the resulting histogram, while containing fewer buckets, is as close to the original one as possible. In general, to decide which buckets to merge, we use a penalty function that returns the cost in histogram accuracy of merging a pair of buckets.

**Calculating Penalties.** Suppose we want to merge two buckets $b_1$ and $b_2$ in a given histogram $H$. Let $H'$ be the resulting histogram after the merge. We define the penalty of merging buckets $b_1$ and $b_2$ in $H$ as follows:

$$penalty(H, b_1, b_2) = \int_{p \in \text{dom}(D)} |\text{est}(H, p) - \text{est}(H', p)| \, dp$$

where $\text{dom}(D)$ is the domain of the data set $D$. In other words, the penalty for merging two buckets measures the difference in approximation accuracy between the old, more expressive histogram where both buckets are separate, and the new, smaller histogram where the two (and perhaps additional buckets) have been collapsed. A merge with a small penalty results in little difference in approximation for range queries and therefore
Fig. 5.9: Parent-child merge to reduce the number of histogram buckets.

will be preferred over another merge with higher penalty. Since the estimated density of tuples inside a bucket is constant by definition, we can calculate penalty functions efficiently. We can identify all regions \( r_i \) in the data domain with uniform density of tuples both before and after the merge, and just add a finite number of terms of the form \( |\text{est}(H, r_i) - \text{est}(H', r_i)| \) to the penalty value. This procedure will become clearer in the rest of this section when we instantiate `merging` operators.

We identified two main kinds of merges for STHoles histograms, which correspond to merging “adjacent” buckets in the tree representation of an STHoles histogram: `parent-child` merges, where a bucket is merged with its parent, and `sibling-sibling` merges, where two buckets with the same parent are merged together (possibly taking some of the parent space). We discuss these operations below.

**Parent-Child Merges:** This operation is useful to eliminate buckets that become too similar to their parents (e.g., buckets with children that cover all interesting regions). Suppose we want to merge buckets \( b_c \) and \( b_p \), where \( b_p \) is \( b_c \)'s parent. After the merge (Fig. 5.9) a new bucket \( b_n \) replaces \( b_p \), and bucket \( b_c \) disappears. The new bucket \( b_n \) has \( \text{box}(b_n) = \text{box}(b_p) \) and \( f_{b_n} = f_{b_c} + f_{b_p} \). The children of both buckets \( b_c \) and \( b_p \) become children of the new bucket \( b_n \). Therefore, we have that \( v(b_n) = v(b_c) + v(b_p) \). The only regions in the original histogram that change the estimated number of tuples after the merge are \( b_p \) and \( b_c \). In conclusion, we have that:

\[
\text{penalty}(H, b_p, b_c) = \left| f_{b_p} - f_{b_n} \frac{v(b_p)}{v(b_n)} \right| + \left| f_{b_c} - f_{b_n} \frac{v(b_c)}{v(b_n)} \right| \frac{|\text{est}(H, b_p) - \text{est}(H', b_p)|}{|\text{est}(H, b_c) - \text{est}(H', b_c)|}
\]
where \( H' \) is the histogram that results from merging \( b_p \) and \( b_c \) in \( H \). The remaining points \( p \) in the histogram domain are such that \( \text{est}(H,p) = \text{est}(H',p) \), so they do not contribute to the merge penalty.

**Sibling-Sibling Merges:** This operation is useful to extrapolate frequency distributions to yet unseen regions in the data domain, and also to consolidate buckets with similar density that cover close regions. Consider the merge of buckets \( b_1 \) and \( b_2 \), with common parent \( b_p \) (Figure 5.10). We first determine the bounding box of the resulting bucket \( b_n \). We define \( \text{box}(b_n) \) as the smallest box that encloses both \( b_1 \) and \( b_2 \) and does not intersect partially with any other child of \( b_p \) (that is, we start with a bounding box that tightly encloses \( b_1 \) and \( b_2 \) and progressively expand it until it does not intersect partially with any other child of \( b_p \)). In the extreme situation that \( \text{box}(b_n) \) is equal to \( b_p \), we transform the sibling-sibling merge of \( b_1 \) and \( b_2 \) into two parent-child merges, namely \( b_1 \) and \( b_p \), and \( b_2 \) and \( b_p \). Otherwise, we define the set \( I \) of “participant” buckets as the set of \( b_p \)'s children (excluding \( b_1 \) and \( b_2 \)) that are included in \( \text{box}(b_n) \). After the merge, the new bucket \( b_n \) replaces buckets \( b_1 \) and \( b_2 \). In general, \( b_n \) also contains a portion of the old \( b_p \). The volume of that part is \( v_{\text{old}} = v_{\text{Box}}(b_n) - (v_{\text{Box}}(b_1) + v_{\text{Box}}(b_2) + \sum_{i \in I} v_{\text{Box}}(b_i)) \).

Therefore, the frequency of the new bucket is \( f_{b_n} = f_{b_1} + f_{b_2} + f_{b_p} \frac{v_{\text{old}}}{v_{b_p}} \). Also, the modified frequency of \( b_p \) in the new histogram becomes \( f_{b_p}(1 - \frac{v_{\text{old}}}{v_{b_p}}) \). To complete the merge, the buckets in \( I \) and the children of the old \( b_1 \) and \( b_2 \) become children of the new \( b_n \). Therefore, we have that \( v(b_p) = v_{\text{old}} + v(b_1) + v(b_2) \). The only regions in the original histogram
that change the estimated number of tuples after the merge are the ones corresponding to \( b_1, b_2 \), and the portion of \( b_p \) enclosed by \( \text{box}(b_n) \). Hence:

\[
\text{penalty}(H, b_1, b_2) = \left| f_{b_n} \cdot \frac{v_{\text{old}}}{v(b_n)} - f_{b_p} \cdot \frac{v_{\text{old}}}{v(b_p)} \right| + \left| f_{b_1} - f_{b_n} \cdot \frac{v(b_1)}{v(b_n)} \right| + \left| f_{b_2} - f_{b_n} \cdot \frac{v(b_2)}{v(b_n)} \right| \]

where \( H' \) is the histogram that results from merging \( b_1 \) and \( b_2 \) in \( H \), and \( r_{\text{old}} \) is the portion of the old bucket \( b_p \) covered by the new bucket \( b_n \). The remaining points \( p \) in the histogram domain satisfy \( \text{est}(H, p) = \text{est}(H', p) \), so they do not contribute to \( \text{penalty}(H, b_1, b_2) \).

Putting all pieces together, Figure 5.1.2.3 shows the refined \( STHoles \) construction algorithm presented at the beginning of this section.

```plaintext
BuildAndRefine (H: \( STHoles \) histogram, q: query plan node)
1  \( T_{b_i} = \text{number of tuples in } R(q) \cap b_i \text{, for each bucket } b_i \in H \) // feedback
2  for each bucket \( b_i \) such that \( R(q) \cap b_i \neq \emptyset \) do
3     \((c_i, T_{c_i}) = \text{Shrink} ( b_i, q, T_{b_i} ) \) // approximate shape if necessary
4     if \( \text{est}(H, c_i) \neq T_{c_i} \) then
5         \text{DrillHole}(b_i, c_i, T_{c_i})
6  while H has too many buckets,
7     merge the pair of buckets in H with the lowest penalty
```

Figure 5.11: Refining an \( STHoles \) histogram using query feedback.

As described so far, we intercept intermediate result streams whenever possible. Alternatively, we can sometimes use sampling to further improve efficiency. For instance, we can gather information from feedback of some, but not all input queries. This decision can be based on the estimated cost of the query, or the quality of the existing statistics used to estimate its execution plan. We can also sample at the tuple level, and gather information over just a random portion of the data stream out of each operator. Then, the overall overhead of the feedback-driven technique is reduced, but the results are still statistically relevant.
5.1.3 Experimental Evaluation

We now report experimental results concerning STHoles histograms. In particular, Section 5.1.3.1 evaluates the performance of STHoles histograms against that of existing techniques for histogram construction. This way, we compare the incremental construction of SITs against an alternative that materializes generating queries and then builds “static” histograms over the temporary results. Section 5.1.3.2 shows some additional experiments that explore specific aspects of STHoles histograms.

For simplicity, we run our experiments over base tables. However, this scenario is general. Building an STHoles histogram over an intermediate result presents no additional challenges: STHoles histograms are refined by just analyzing result streams, independently from the operator that produces such streams. We now define the data sets, histograms, and workloads used in the experiments.

Data Sets

We use both synthetic and real data sets for the experiments. The real data sets we consider [BM98] are: Census2D and Census3D (two- and three-dimensional projections of a fragment of US Census Bureau data) consisting of 210,138 tuples, and Cover4D (four-dimensional projection of the CovType database, used for predicting forest cover types from cartographic variables), consisting of 545,424 tuples. We also generated synthetic data sets for our experiments following different data distributions, as described below.

- **Gauss**: The Gauss synthetic distributions [PFTV93] consist of a predetermined number of overlapping multidimensional Gaussian bells. The parameters for these data sets are: the number of Gaussian bells \( p \), the standard deviation of each peak \( \sigma \), and a Zipfian parameter \( z \) that regulates the total number of tuples contained in each Gaussian bell.

- **Array**: These data sets were used in [AC99]. Each dimension has \( v \) distinct values, and the value sets of each dimension are generated independently. Frequencies are
CHAPTER 5. BUILDING SITS

### Data Set

<table>
<thead>
<tr>
<th>Attribute</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>d: Dimensionality</td>
<td>2</td>
</tr>
<tr>
<td>N: Cardinality</td>
<td>500,000</td>
</tr>
<tr>
<td>R: Data domain</td>
<td>([0 \ldots 1000]^d)</td>
</tr>
<tr>
<td>z: Skew</td>
<td>1</td>
</tr>
<tr>
<td>Gauss</td>
<td></td>
</tr>
<tr>
<td>p: Number of peaks</td>
<td>100</td>
</tr>
<tr>
<td>σ: Peaks’ standard deviation</td>
<td>25</td>
</tr>
<tr>
<td>Array</td>
<td></td>
</tr>
<tr>
<td>v: Distinct attribute values</td>
<td>100</td>
</tr>
</tbody>
</table>

Figure 5.12: Default values for the synthetic data sets used in the STHoles experimental evaluation.

generated according to a Zipfian distribution and assigned to randomly chosen cells in the joint frequency distribution matrix. The parameters for this data set are the number of distinct attributes by dimension \(v\), and the Zipfian value for the frequencies \(z\). When all the data points are equidistant, this data set can be seen as an instance of the Gauss data set with \(σ = 0\) and \(p = v^d\).

Figure 5.12 summarizes the default values for the synthetic data set parameters.

**Histograms**

We compare our STHoles histograms against the following multidimensional histograms: *EquiDepth* [MD88], *MHist* [PI97], *STGrid* [AC99], and *GenHist* [GKTD00], using the values of parameters that the respective authors considered the best. All experiments allocate the same amount of memory for all histograms techniques, which however translates to different numbers of buckets for each. Consider the space requirements for \(B\) \(d\)-dimensional buckets. Both *EquiDepth* and *MHist* histograms require \(2 \cdot d \cdot B\) values for the bucket boundaries plus \(B\) frequency values. *STGrid* histograms need \(B\) values for frequencies plus around \(\sqrt[4]{B}\) values for the unidimensional rulers [AC99]. *GenHist* histograms require \(2 \cdot B\) values for bucket positions plus \(B\) frequency values. Finally,
STHoles histograms use $2 \cdot d \cdot B$ values for bucket boundaries, $B$ values for frequencies, and $2 \cdot B$ pointers for maintaining the tree structure, since each bucket needs to point to its “first” child plus a sibling. By default, the available memory for a histogram is fixed to 1,000 bytes.

**Workloads**

We use a slightly modified version of the framework given in [PSTW93] to generate probabilistic models for range queries. Given a data set, a range query model is defined as a pair \((C, R[v])\), where $C$ is the distribution of the query centers, $R$ is a function that constrains the query boundaries, and $v$ is a constant value for function $R$. To obtain a workload given a query model, we first generate the query centers using $C$ and then expand their boundaries so they follow $R[v]$. For our experiments, we consider the following center distributions, which are considered representative of user behavior (see [PSTW93]):

- **Data**: The query centers follow the data distribution.
- **Uniform**: The query centers are uniformly distributed in the data domain.
- **Gauss**: The query centers follow a Gauss distribution independent of the data distribution.

The range constraints we used for our experiments are:

- **V[c_v]**: The range queries are hyper-rectangles included in a hypercube of volume $c_v$, and model the cases in which the user specifies the query values in terms of a window area.
- **T[c_t]**: The range queries are hyper-rectangles that cover a region with $c_t$ tuples, and model the situations in which the user has knowledge about the data distribution and issues queries with the intention of retrieving a given number of tuples.
Parameters $c_v$ and $c_t$ are specified as a percentage of the total volume and number of tuples of the data distribution, respectively.

By combining these parameters we obtain six different probabilistic models for query workloads. By default, we use 1% for both $c_v$ and $c_t$. As an example, the query model $\langle Data, T[1\%] \rangle$ results in queries with centers that follow the data distribution and contain 1% of the tuples in the data set. Similarly, the query model $\langle Gauss, V[1\%] \rangle$ corresponds to queries with centers that follow a multi-gaussian distribution and have an average volume of around 1% of the data domain. Figure 5.13 shows two sample workloads of 50 queries each for the Census2D data set.

**Metrics**

To compare our new technique against existing ones, we first construct a training workload that consists of 1,000 queries and use it to tune the STHoles and STGrid histograms. Then, we generate a validation workload from the same distribution as the training workload that also consists of 1,000 queries, and calculate the average absolute error for all the histograms. Given a data set $D$, a histogram $H$, and a validation workload $W$, the **average absolute error** $E(D, H, W)$ is calculated as follows:

$$E(D, H, W) = \frac{1}{|W|} \sum_{q \in W} |est(H, q) - act(D, q)|$$

where $est(H, q)$ is the estimate of the number of tuples in the result of $q$, using histogram $H$ for the estimation, and $act(D, q)$ is the actual number of $D$ tuples in the result of $q$.

![Figure 5.13: Two workloads for the Census2D data set.](image)

(a) Census2D data set. (b) $\langle Data, T[1\%] \rangle$ workload. (c) $\langle Gauss, V[1\%] \rangle$ workload.

Figure 5.13: Two workloads for the Census2D data set.
We choose average absolute errors as the accuracy metric, since relative errors tend to be less robust when the actual number of tuples for some queries is zero or near zero. In general, however, absolute errors greatly vary across data sets, making it difficult to report results for different data sets. Therefore, for each experiment, we normalize the average absolute error by dividing it by \( E_{unif}(D, W) = \frac{1}{|W|} \sum_{q \in W} |est_{unif}(D, q) - act(D, q)| \), where \( est_{unif}(D, q) \) is the result size estimate obtained by assuming uniformity, i.e., in the case where no histograms are available. We refer to the resulting metric as Normalized Absolute Error.

### 5.1.3.1 Comparison of STHoles and other Histogram Techniques

**Accuracy of Histograms.** Figure 5.14 shows normalized absolute errors for different histograms, data sets and workloads. We can see from the figures that STHoles histograms give better results than EquiDepth, MHist, and STGrid in virtually all cases. On the other hand, STHoles and GenHist are comparable in accuracy, and although STHoles histograms do not directly inspect the data distributions, in many cases they outperform GenHist histograms. The only data set in which GenHist results in significantly better accuracy than STHoles is Cover4D (see Figure 5.14). For this high-dimensional data set, the ability to capture interesting data patterns based only on workload information is diminished. However, it is important to note that, even for high dimensions, STHoles histograms still produce better results than do MHist, EquiDepth, and STGrid histograms. GenHist has a high error rate of 75% for the Array data set with the \( \langle Data, T[1\%] \rangle \) workload. This may be due to the choice of histogram construction parameter values in [GKTD00], which is independent of the underlying data set. In general, note that STHoles, GenHist and, to a limited extent, EquiDepth histograms are “robust” across different data sets and workloads, in the sense that they consistently produce reasonable results. In contrast, STGrid and MHist become too inaccurate for some combinations of data sets and workloads.
CHAPTER 5. BUILDING SITS

Figure 5.14: Normalized absolute error for different histograms, data sets, and workloads.
Figure 5.15: Normalized absolute error using $\langle Data, T[c_v]\rangle$ for varying spatial ($c_v$) and tuple ($c_t$) selectivity.
Robustness across Workloads. Figure 5.15 shows the normalized absolute error for different data sets and for varying selectivity $s$ for workloads $(Data, V[s])$ and $(Data, T[s])$, respectively. We can see that in almost all cases STHoles histograms outperform traditional techniques. Even in the few cases that STHoles histograms are not the most accurate, they are a close second, with only one exception. Our technique is not too accurate in Figure 5.15(d) for tuple selectivity $c_t = 0.1\%$ (and neither are MHist and STGrid). This is mainly because in the Gauss data set the $(Data, T[0.1\%])$ workload consists of many small and disjoint queries. This workload is inevitably bad for any histogram refinement technique like STHoles that bases all decisions on query feedback, without examining the actual data sets at any time. If such workloads are expected, we can start the construction of the STHoles histograms with a more informed representation of the data set. In particular, we can use an existing histogram over a base table (therefore making some simplifying assumptions) as the starting point for our technique in the algorithm of Section 5.1.2. We also varied data set skew and dimensionality for the synthetic data sets. The results we obtained are similar to those we have presented so far and are discussed in [BCG01b].

In conclusion, although for some particular configurations STHoles histograms are slightly outperformed by others (notably in one data point of Figure 5.15(d)), in general STHoles is a stable technique across different workloads and data sets, and in many cases results in significantly lower estimation errors than multidimensional histograms that inspect the data sets.

5.1.3.2 Experiments Specific to Histogram Refinement

Convergence. Our techniques for building STHoles histograms keep adjusting the histograms as queries are evaluated. We now study how the quality of the STHoles histograms varies with the number of observed queries. To do so, we train the STHoles histogram 50 queries at a time, and after each step we calculate the normalized absolute error using the complete validation workload. Figure 5.16 shows the results for different
data sets and workloads. We can see that STHoles histograms converge fairly quickly, and generally need only around 150-200 queries to get stable results.

**Effect of Updates.** Data sets are rarely static, and the data distribution might change over time. We now evaluate how well our new techniques adapts to changing data distributions. For this, we start with the Array and Gauss data sets, and progressively “morph” one into the other using random tuple swaps. Each column of four points in Figure 5.17 represents a different experiment where we vary the percentage of tuples that
are swapped between the two data sets. For instance, in Figure 5.17(a) we start with the
Array data set. We build the static GenHist, MHist and EquiDepth histograms using
this data set, and train the STHoles and STGrid histograms using the first half of a
\langle Data, V[1]\% \rangle workload. Then, we randomly select a percentage of tuples from the origi-
nal Array data set and interchange them with randomly selected tuples from the Gauss
data set. After that, we finish the training of the STHoles and STGrid histograms using
the remaining half of the workload. Finally, we test all histograms using a validation
\langle Data, V[1]\% \rangle workload. Analogously, Figure 5.17(b) shows the results when starting
with a Gauss data set and changing it to an Array data set.

Figure 5.17: Normalized absolute error after updates.
CHAPTER 5. BUILDING SITS

Not surprisingly, we can see that the static histograms become really inaccurate when the underlying data distribution changes. In some cases the results are even worse than when assuming uniformity and independence, which highlights that periodically rebuilding such multidimensional histograms is essential (we include in the plots these static histograms just to quantify this behavior). In contrast, both STGrid and STHoles adapt gracefully to changes in the data distribution. For STHoles histograms we observe almost no degradation even when changing the data set completely. That is not the case for STGrid histograms. For instance, in Figure 5.17(b) we can see that STHoles keeps the error rate below 17% at all times, while STGrid results in over 67% of normalized absolute error for 100% tuple interchanges.

5.2 An Off-line Approach to Build SITs

In the previous section we showed a technique that uses feedback from the execution of user queries to build and refine SITs on the fly with small overhead. When appropriate queries are posed to the RDBMS, this technique can build accurate SITs without looking at the actual data, therefore avoiding the evaluation of the generating queries of the SITs. We showed experimentally that STHoles SITs converge relatively quickly to a very accurate representation of the underlying data distribution.

While this approach is useful in some situations, the creation and refinement of histograms on the fly is not applicable in other scenarios. For instance, a SIT that could be highly beneficial for a common query might never we materialized this way if the (sub-optimal) plan of choice for the query does not include the SIT’s generating query as a subexpression. In such a case, the SIT’s generating query is not executed and hence relevant query feedback is not produced. Figure 5.18 shows an example of this scenario, where the optimizer rejects the (optimal) plan on the left because the intermediate result of sub-plan \((\sigma_{R.a<10}(R) \bowtie S)\) is estimated to be large. If SIT\((R.a|R \bowtie S)\) were present in the system, the cardinality of \(\sigma_{R.a<10}(R \bowtie S)\) would have been correctly estimated and the optimizer would have chosen the plan on the left. Instead, based on the available
SITs, the optimizer chooses the alternative plan on the right. Unfortunately, such plan cannot be exploited to refine the useful SIT($R.a | R \bowtie S$). Therefore, subsequent queries will continue to choose the same sub-optimal plan at the right of Figure 5.18. Moreover, even if the required SITs may be built using feedback from the executed queries, the database administrator might want to create some helpful SITs at once without waiting for STHoles SITs to converge. Finally, the approach in the previous section restricts the structure of SITs to statistical summaries that can be built and refined using query feedback, such as STHoles histograms. If the RDBMS does not support such statistics, the feedback-based approach described earlier cannot be applied.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure5.18.png}
\caption{Useful SITs cannot always be extracted from execution plans.}
\end{figure}

To address the scenarios described above, we now introduce a complementary family of techniques to efficiently create SITs. The procedures in this section are off-line and balance efficiency of construction and accuracy of the resulting SITs. Specifically, we create SIT($A|Q$) by effectively working on a (close) approximation of generating query $Q$, which is obtained \textit{without executing} $Q$. The key technical difficulty is how to define such approximations. If we use an approximation that is not close to the actual distribution of the generating query, most of the benefits of using SITs will probably be lost. The key idea used in this section is to \textit{approximate} the execution of query expressions by relying on some (but not all) of the simplifying assumptions that are used during cardinality estimation. On one extreme, if we rely on \textit{every} possible simplifying assumption, the resulting SITs would carry the same information as base-table statistics, and will therefore be useless
during cardinality estimation. On the other extreme, if we do not rely on any assumption whatsoever, the resulting SITs would be as accurate (but also as expensive to build) as the result of the corresponding query expressions. In this section we introduce techniques that lie in between these two extreme approaches.

In Section 5.2.1 we briefly review the main assumptions used during cardinality estimation of SPJ queries. Next, in Section 5.2.2 we introduce Sweep, a family of techniques to create SITs that balance accuracy and efficiency of construction. We then recognize that while trying to create several SITs at once, commonalities between their generating queries can be leveraged. In Section 5.2.3 we formalize the problem of optimally creating a set of SITs by interleaving individual executions of Sweep. Finally, in Section 5.2.4 we report an experimental evaluation of the techniques of Sections 5.2.2 and 5.2.3.

5.2.1 Review of the Main Assumptions behind Cardinality Estimation

Recall from Chapter 3 the main simplifying assumptions used for estimating the cardinality of the following SPJ query:

\[
\text{SELECT * FROM } R, S \\
\text{WHERE } R.x=S.y \text{ and } S.a<10
\]

**Independence Assumption:** When propagating histogram \(H(S,a)\) through the predicate \(R \bowtie_{x=y} S\), the implicit assumption is that the distributions of attributes in \(R\) and \(S\) are independent. For this reason, all bucket frequencies of histogram \(H(S,a)\) are multiplied by the same scaling factor.

**Containment Assumption:** The contents of the histograms used to estimate \(R \bowtie S\) are modelled as if \(R \bowtie S\) were close to a foreign-key join.

**Sampling Assumption:** When base-table histograms are built using sampling, the number of distinct values per bucket can be difficult to approximate. The sampling assumption dictates that such approximations are accurate estimators of the correct values.
CHAPTER 5. BUILDING SITS

As discussed in Section 3.1, the independence assumption is the main source of errors during cardinality estimation. For that reason, we do not rely on such assumption to approximate SITs. In contrast, the containment and sampling assumptions usually contribute only moderately to errors in cardinality estimates. In the next section we introduce Sweep, a technique that efficiently approximates SITs by relying on the containment and sampling assumptions.

5.2.2 Creating a Single SIT

As discussed earlier, if we build a given SIT over an approximate query expression, we can obtain a good balance between accuracy and efficiency of construction. In Section 5.2.2.1 we introduce Sweep, a technique to efficiently create SITs with two-way join generating queries. Sweep efficiently builds accurate approximations of SITs by relying on the containment and sampling assumptions but it does not rely on the independence assumption. Then, in Section 5.2.2.2 we present some alternatives that are more accurate, and at the same time less efficient, than the original formulation of Sweep. Finally, in Sections 5.2.2.3 and 5.2.2.4 we show how to extend Sweep to deal with more complex generating queries.

5.2.2.1 Architecture of Sweep

To create SIT(R,a|Q), Sweep attempts to efficiently generate a sample of $\pi_{R,a}(Q)$ without actually executing $Q$, and then use existing techniques for building histograms over this intermediate sample. We now describe Sweep by using an example.

Suppose that we want to create SIT(S,a|R $\bowtie_{x=y}$ S), shown in Figure 5.19. In Step 1, we perform a sequential scan\(^1\) over table $S$. For each tuple $(y_i, a_i)$ scanned from $S$, in Step 2 we estimate the number of tuples $r \in R$ such that $r.x = y_i$. We denote this value the multiplicity of tuple $y_i$ from table $S$ in table $R$. (The component used to calculate multiplicity values, called $m$-Oracle, is explained later in this section.) Clearly,

\(^1\)If an index over attributes $\{S.y, S.a\}$ is available, we replace the sequential scan by an appropriate index scan.
CHAPTER 5. BUILDING SITS

the multiplicity of \( y_i \) in \( R \) is the number of matches for tuple \((y_i, a_i)\) in the join \( R \bowtie_{x=y} S \).

If the estimated number of matches for \( y_i \) is \( n \), we append \( n \) copies of \( a_i \) to a temporary table in Step 3. It is important to note that we do not materialize the temporary table, but we treat it as a stream of values that approximate \( \pi_{S,a}(R \bowtie_{x=y} S) \). Then, in Step 4, we apply a one-pass sampling algorithm over the streaming table, such as the reservoir sampling technique of [Vit85] or the techniques in [GM98] for concise samples. The result of this step is an approximate sample of \( \pi_{S,a}(R \bowtie_{x=y} S) \), which is precisely the distribution over which we want to build the SIT. Finally, in Step 5, we use a traditional histogram technique to build \( \text{SIT}(S.a|R \bowtie_{x=y} S) \) from the sample.

Figure 5.19: Creating \( \text{SIT}(S.a|R \bowtie_{R.x=S.y} S) \) using \textit{Sweep}.
Approximating Multiplicity Values

To create $\text{SIT}(S.a \mid R \bowtie_{x=y} S)$ using Sweep we need to obtain, both efficiently and accurately, the multiplicity of arbitrary tuples from $S$ in $R$ (Step 2 in Figure 5.19). We now introduce a histogram-based technique that results in accurate and efficient approximations of multiplicity values. The idea is to use histograms $H(R.x)$ and $H(S.y)$ to provide the multiplicity of tuples from $S$ in $R$. For a given value $y$ from $S$, we first identify the buckets $b_{R,y}$ and $b_{S,y}$ from histograms $H(R.x)$ and $H(S.y)$ that contain value $y$. Then, we calculate the expected number of tuples from $R$ that join with $y$ under the containment assumption for join predicates.

To estimate the multiplicity of $y$ in $R$ we consider two scenarios. Let $dv_{R,y}$ and $dv_{S,y}$ be the number of distinct values in buckets $b_{R,y}$ and $b_{S,y}$, respectively. If the number of distinct values from $H(R.x)$ is larger than the number of distinct values from $H(S.y)$ (i.e., $dv_{S,y} \leq dv_{R,y}$), the containment assumption indicates that value $y$ – which belongs to one of the $dv_{S,y}$ groups in $b_{S,y}$ – matches some of the $dv_{R,y}$ groups in $b_{R,y}$. Since we use the uniformity assumption inside buckets, the multiplicity for $y$ in this situation would be $f_{R,y}/dv_{R,y}$, where $f_{R,y}$ is the frequency of bucket $b_{R,y}$. However, if $dv_{S,y} > dv_{R,y}$, we can no longer guarantee that value $y$ joins with some of the $dv_{R,y}$ buckets in $H(R.x)$. If the tuples that satisfy the join are distributed uniformly, the probability that tuple $y$ belongs to one of the $dv_{R,y} < dv_{S,y}$ groups in $b_{R,y}$ that match some group in $b_{S,y}$ is $dv_{R,y}/dv_{S,y}$. In that case, the multiplicity would be $f_{R,y}/dv_{R,y}$. Otherwise $y$ does not match with any value in $R$ and the multiplicity would be 0. In conclusion, when $dv_{S,y} > dv_{R,y}$, the expected multiplicity of $y$ in $R$ is $(f_{R,y}/dv_{R,y}) \cdot (dv_{R,y}/dv_{S,y}) + 0 \cdot (1 - dv_{R,y}/dv_{S,y}) = f_{R,y}/dv_{S,y}$.

Putting both results together, the expected multiplicity of $y$ from $S$ in $R$ is given by $f_{R,y}/\max(dv_{R,y}, dv_{S,y})$. Since we can locate the bucket that contains a given tuple efficiently in main memory, this procedure is inexpensive. In Section 5.2.4 we show experimentally that the technique explained above is also an accurate estimator of multiplicity values.
5.2.2.2 Balancing Accuracy and Efficiency

When creating SIT(S.a|R \bowtie_{x=y} S), Sweep avoids relying on the independence assumption by scanning the multidimensional distribution (S.a, S.x) in Step 1. However, Sweep does rely on the containment assumption (when estimating multiplicity values in Step 2), and on the sampling assumption (when building SITs by using the samples obtained in Step 4). We now describe how to avoid relying on these assumptions as well, and therefore obtain more accurate SITs. The resulting procedures, however, are more expensive than the original Sweep.

- **SweepIndex** (Containment assumption): If an index over attribute R.x is available in Step 2, we can perform repeated index lookups to find exact multiplicity values. Since index lookups generally take more time than histogram lookups, SweepIndex is more expensive than Sweep.

- **SweepFull** (Sampling assumption): We can omit Step 4 in Sweep and build the SIT directly from the temporary table. Of course, this might require materializing the temporary table on disk if the table becomes too large to fit in main memory. For that reason, SweepFull is more expensive than Sweep.

- **SweepExact** (Containment and sampling assumptions): By combining the ideas used in both SweepIndex and SweepFull we avoid relying on any assumption whatsoever, and the result of SweepExact is the same as if we evaluate the SIT’s query and then build a histogram over the result. This technique returns the most accurate histogram for a given SIT, but is also the most expensive.

In the rest of the section, we focus on the original formulation of Sweep and we extend it to handle other generating queries. Later, in Section 5.2.4, we compare the accuracy of the alternatives introduced in this section.
5.2.2.3 Beyond Two-way Join Generating Queries

The basic formulation of \textit{Sweep} handles SITs with two-way join generating queries. In this section we study how we can extend \textit{Sweep} to handle queries with acyclic join graphs. (The \textit{join graph} of query \(q\) has one node per table in \(q\) and one edge between two nodes if the corresponding tables are joined in \(q\).) Specifically, we focus on acyclic join queries for which every pair of tables \(t_1\) and \(t_2\) in the generating query \(q\) is joined by at most one predicate. In the general case (e.g., for \(R \bowtie_{w=R.x \land y=R.y} S\)), we can still apply the techniques in this section by using multidimensional histograms (see also Section 5.2.2.4).

Linear-join Generating Queries

We first extend \textit{Sweep} to handle linear-join generating queries. A linear-join query can be expressed as \((R_1 \bowtie \ldots \bowtie R_n)\), where the \(i\)-th join (\(1 \leq i \leq n-1\)) connects tables \(R_i\) and \(R_{i+1}\). In other words, the join graph of a linear-join query is a chain of nodes. To approximate \(\text{SIT}(S.a|R \bowtie_{x=y} S)\) we use the following information (see Section 5.2.2.1):

- Attributes \(S.y\) and \(S.a\) in table \(S\) (accessed through a sequential scan)
- Histograms \(H(R.x)\) and \(H(S.y)\) (to approximate multiplicity values).

To approximate a SIT over a linear-join query, we associate the query joins and unfold the original SIT into a sequence of applications of \textit{Sweep}, as illustrated below.

\textbf{Example 18} Consider \(\text{SIT}(U.a|R \bowtie_{r_1=s_1} S \bowtie_{s_2=t_1} T \bowtie_{t_2=u_1} U)\), which is shown in Figure 5.20. We can rewrite such SIT as \(\text{SIT}(U.a|RST \bowtie_{t_2=u_1} U)\), where \(RST\) is defined as \(R \bowtie_{r_1=s_1} S \bowtie_{s_2=t_1} T\). To approximate this equivalent SIT, we need to perform a sequential scan over table \(U\), and access histograms \(H(U.u1)\) and \(H(RST.t2)\). However, \(H(RST.t2)\) is just \(\text{SIT}(T.t2|R \bowtie_{r_1=s_1} S \bowtie_{s_2=t_1} T)\). We then recursively create \(\text{SIT}(T.t2|R \bowtie_{r_1=s_1} S \bowtie_{s_2=t_1} T)\), which in turn requires creating \(\text{SIT}(S.s2|R \bowtie_{r_1=s_1} S)\). The last SIT corresponds to the base case, since both of the required histograms are specified over base tables. \(\blacksquare\)
Acyclic-Join Generating Queries

To handle generating queries with general acyclic join graphs, we proceed as follows. First, we convert the join graph into the tree that is rooted at the table holding the SIT’s attribute. As an example, Figure 5.21(a) shows a SIT whose generating query has an acyclic join graph, and Figure 5.21(b) shows the corresponding tree for attribute U.a.

Suppose that the height of the tree is one (i.e., the tree consists of root node R and children S1, . . . , Sn), and we want to get SIT(R.a|R ≪R.r1=S1.s1 S1 ≪S2.s2 S2 ≪ . . . ≪R,rn=Sn.sn Sn). In this case we can extend Sweep as follows. We first obtain base-table histograms for each attribute Si.sj and, similar to the original formulation of Sweep, we perform a sequential scan over R. To obtain the multiplicity of tuple r = (a, r1, . . . , rn)
from $R$ in the multi-way join, we first get $m_i$, the partial multiplicity of $r_i$ in $S_i$, for all $i$. Then, we calculate the multiplicity of tuple $r$ from $R$ in the join as $\Pi_i m_i$. We note that this multiplicity value is obtained without assuming independence between join predicates. In fact, each tuple $r$ from $R$ joins with $m_i$ tuples from $S_i$, and since the join graph is acyclic, the final result contains all possible combinations between $r$ and each qualifying tuple in $S_i$. After obtaining the multiplicity values, we proceed with Sweep as usual.

For an arbitrary acyclic-join generating query, we combine the procedure for linear-join queries and the technique described above for tree joins of height one. Essentially, we perform a post-order traversal of the join tree. At each leaf node $n$ we obtain the base-table histograms of the attributes that participate in the join between node $n$ and its parent node in the join tree. At each internal node $n$, we obtain previous SITs (or base-table histograms) for each child of $n$, and use them to compute the SIT for the attribute in $n$ that participates in the join predicate with $n$’s parent (or the attribute of the original SIT if $n$ is the root of the join tree). We illustrate this procedure in the example below.

**Example 19** Consider the SIT shown in Figure 5.21(a), and its corresponding join tree in Figure 5.21(b). To create $\text{SIT}(U.a | R \bowtie T \bowtie V \bowtie U \bowtie S)$, we traverse the join tree in post-order, i.e., $[S, R, V, T, U]$. We first obtain base-table histograms $H(S.s1)$, $H(R.r1)$, and $H(V.v1)$. We then process node $T$ and create $\text{SIT}(T.t3 | R \bowtie r_1 = t_1 \bowtie T \bowtie t_2 = v_1 \bowtie V)$. For that purpose, we perform a sequential scan over table $T$ and exploit histograms $H(R.r1)$ and $H(V.v1)$ and $H(T.t1)$. Finally, we process node $U$ and calculate the desired $\text{SIT}(U.a | R \bowtie r_1 = t_1 \bowtie T \bowtie t_2 = v_1 \bowtie V \bowtie t_3 = u_1 \bowtie U \bowtie u_2 = s_1 \bowtie S)$ using the base-table histogram $H(S.s1)$ and the SIT produced at node $T$.

### 5.2.2.4 Other Extensions

We now briefly discuss further extensions for Sweep to handle other types of generating queries and, in some cases, multidimensional SITs.
CHAPTER 5. BUILDING SITS

Selections in the SIT Generating Queries

We can often easily extend the procedures in Section 5.2.2.3 to handle generating queries with selection predicates. As an example, consider \( \text{SIT}(S.a | \sigma_{S.b < 5}(R \bowtie S)) \). Since the selection predicate \( S.b < 5 \) is on table \( S \), we can simply filter tuples that satisfy \( S.b < 5 \) during the scan of table \( S \). (We can use a clustered index over \( S.b \) if available, thus improving execution time.) In general, we can extend this idea for more complex generating queries. For instance, to obtain \( \text{SIT}(R.a | \sigma_{S.a = 2}((R .t_{w-x} S) \bowtie_{y-z} T)) \) we perform a sequential scan over table \( R \) (keeping tuples that satisfy \( R.c < 5 \)), and manipulate histograms \( \text{SIT}(R.w | \sigma_{R.c < 5}(R)) \) and \( \text{SIT}(S.x | \sigma_{S.b = 2}(S \bowtie_{y-z} T)) \). The former SIT can be created using an additional scan (or a sample) over table \( R \). The latter is recursively obtained with a sequential scan over \( S \) (keeping tuples satisfying \( S.b = 2 \)) and manipulating histograms \( \text{SIT}(S.y | S.b = 2) \) and base-table histogram \( H(T.z) \). If the filter predicate is defined over a table that is not scanned during \( \text{Sweep} \), we can still obtain the corresponding SIT by exploiting multidimensional histograms. For instance, to obtain \( \text{SIT}(S.a | \sigma_{R,b < 10}(R \bowtie_{x=y} S)) \) we perform a sequential scan over \( S \) and obtain multiplicity values using a histogram over \( S.y \) and a two-dimensional histogram over \( \{R.b, R.x\} \).

Multidimensional SITs

We can also extend \( \text{Sweep} \) to generate some types of multidimensional SITs. For instance, we create \( \text{SIT}(S.a, S.b | R \bowtie_{x=y} S) \) by first performing a scan over the joint distribution \((S.a, S.b, S.y)\). Then, we obtain multiplicity values and build (and then sample) a temporary table approximating \( \pi_{S.a, S.b}(R \bowtie_{x=y} S) \). Finally, we use a traditional multidimensional technique to materialize the SIT over the approximate sample. In general, we can extend \( \text{Sweep} \) in this manner for SITs of the form \( \text{SIT}(a_1, \ldots, a_k | Q) \), where all the \( a_i \) attributes belong to the same base table. These cases require more space for samples because each element in the sample is a multidimensional tuple over attributes \( \{a_1, \ldots, a_k\} \).
5.2.2.5 Summary

In this section we introduced *Sweep*, a technique to create SITs with acyclic-join generating queries. *Sweep* requires a sequential scan over each involved table (except for the root of the join tree) and some additional processing to materialize intermediate SITs. In the following section we analyze the scenario in which multiple SITs need to be created at once, and explore evaluation strategies for *Sweep* that leverage commonalities among the SIT generating queries.

5.2.3 Creating Multiple SITs

The previous section introduced techniques based on *Sweep* to create a single SIT. However, we are typically interested in creating several SITs at once. For instance, the techniques in Chapter 6 recommend a set of candidate SITs to create for a given workload. In such situations, many commonalities between SITs might exist, and we could share resources (e.g., sequential scans) to build multiple SITs. As we will see, the obvious one-at-a-time approach for building SITs is often suboptimal, as illustrated below.

**Example 20** Consider the following two SITs:

\[
\text{SIT}(T.a | R \bowtie_{r_1=s_1} S \bowtie_{s_3=t_3} T)
\]

\[
\text{SIT}(S.b | R \bowtie_{r_2=s_2} S)
\]

A naive approach to materialize both SITs is to apply *Sweep* to each SIT separately. This would result in one sequential scan over tables \(S\) and \(T\) (to build the first SIT), and a second sequential scan over table \(S\) (to build the second SIT). A more efficient approach scans table \(S\) only once, as described next:

1. Perform a sequential scan over table \(S\) to obtain both SIT\((S.b | R \bowtie_{r_2=s_2} S)\) and SIT\((S.s3 | R \bowtie_{r_1=s_1} S)\). In this case, we share the sequential scan over attributes \(S.s2, S.b, S.s3, \text{and } S.s1\), and use histograms \(H(R.r2)\) and \(H(S.s2)\), and \(H(R.r1)\) and \(H(S.s1)\) to obtain the multiplicity values of the first and second SIT, respectively.
2. Perform a sequential scan over $T$; using histogram $H(T,t_3)$ and the previously calculated $\text{SIT}(S,s_3|R \bowtie_{r=1..s_1} S)$, obtain $\text{SIT}(T,a|R \bowtie_{r=1..s_1} S \bowtie_{s_3=t_3} T)$.

The strategy outlined above requires a single sequential scan over table $S$, and therefore is more efficient than the naive strategy. On the other hand, the memory requirements for the second strategy are larger, since we maintain two sets of samples in memory: one for $\pi_{S,b}(R \bowtie_{r=2..s_2} S)$, and another for $\pi_{S,s_1}(R \bowtie_{r=1..s_1} S)$.

The tradeoffs illustrated in the example above have similar counterparts in multi-query optimization [RSSB00], which aims to exploit common subexpressions among input queries to reduce the combined execution cost. However, unlike traditional multi-query optimization, the execution plan for each individual SIT in our scenario is known in advance (it can be one of few fixed sequences of applications of $\text{Sweep}$ described in Section 5.2.2.3). The execution search space is much smaller than in the general case, and we can provide a tailored solution to this constrained problem. We now state the optimization problem to create multiple SITs:

Given a set of SITs $\mathcal{H} = \{H_1, \ldots, H_n\}$, a sampling rate $s$, and the amount of available memory $M$, find the optimal sequence of applications of $\text{Sweep}$ (sharing sequential scans as explained above) such that:

1. The total memory used for sampling is always bounded by $M$, and
2. The estimated execution time for building $\mathcal{H}$ is minimized.

As we will see, this optimization problem can be viewed as a generalization of the Shortest Common Supersequence problem, or SCS. In Section 5.2.3.1 we review the SCS problem and in Section 5.2.3.2 we discuss a previously proposed technique to solve this problem. Then, in Section 5.2.3.3 we show how to map the SIT creation problem to a generalized version of SCS.

$^2$The sampling rate can be specified as a percentage of the table size, an absolute amount, or a combination or both (depending of the table size).
5.2.3.1 Shortest Common Supersequence

The Shortest Common Supersequence [Mai78] is a well known problem that arises in many applications, such as text editing, data compression, and robot assembly lines. We define the Shortest Common Supersequence problem next.

**Definition 8** Let \( R = x_1 \ldots x_n \) be a sequence of elements (individual elements of \( R \) can be accessed using array notation, so \( R[i] = x_i \)). Given a pair of sequences \( R \) and \( R' \), we say that \( R' \) is a subsequence of \( R \) —and that \( R \) is a supersequence of \( R' \)— if \( R' \) can be obtained by deleting zero or more elements from \( R \). A sequence \( R \) is a common supersequence of a set of sequences \( \mathcal{R} = \{R_1, \ldots, R_n\} \) if \( R \) is a supersequence of each \( R_i \in \mathcal{R} \). A shortest common supersequence of \( \mathcal{R} \), denoted \( \text{SCS}(\mathcal{R}) \), is a common supersequence of \( \mathcal{R} \) with minimal length.

**Example 21** Consider \( \mathcal{R} = \{abdc, bca\} \). Some common supersequences of \( \mathcal{R} \) are \( abdcba \), \( abdcbca \), and \( aabbddeecbbcaa \). In particular, we can prove that \( abdca \) is \( \text{SCS}(\mathcal{R}) \) because no sequence of size four is a common supersequence of both \( abdc \) and \( bca \). 

Finding shortest common supersequences is an NP-complete problem. It can be solved using dynamic programming in \( O(l^n) \) time for \( n \) sequences of length at most \( l \). The idea is to formulate SCS as a shortest path problem in an acyclic directed graph with \( O(l^n) \) nodes [Tim90]. For a given set of sequences \( \mathcal{R} = \{R_1, \ldots, R_n\} \), the graph is constructed as follows. Each node in the graph is an \( n \)-tuple \((r_1, \ldots, r_n)\), where \( r_i \in \{0 \ldots |R_i|\} \) indexes a position in \( R_i \). Node \((r_1, \ldots, r_n)\) encodes a solution for the common supersequence of \( \{S_1, \ldots, S_n\} \), where \( S_i = R_i[1]R_i[2]\ldots R_i[r_i] \) is the \( r_i \)-prefix of \( R_i \) (if \( r_i = 0 \), \( S_i \) is the empty sequence). There is an edge from node \((u_1, \ldots, u_n)\) to node \((v_1, \ldots, v_n)\) with label \( \omega \) if the following properties hold:

1. \( u_i = v_i \) or \( u_i + 1 = v_i \),
2. At least one position \( u_j \) satisfies \( u_j + 1 = v_j \), and
3. For every position \( v_j \) such that \( u_j + 1 = v_j \), we have that \( R_j[v_j] = \omega \).
Informally, an edge labelled ω connects nodes u and v if we can reach the state represented by v from the state represented by u by adding ω to the common supersequence encoded at u. Figure 5.22 shows the graph induced by sequences abdc and bca.

It can be shown [Tim90] that any path from node O = (0, . . . , 0) to node F = (|R1|, . . . , |Rn|) in the graph corresponds to a common supersequence of \( \mathcal{R} \). In particular, any shortest path from O to F corresponds to a shortest common supersequence of \( \mathcal{R} \). Therefore, to solve SCS we can use any known algorithm to find the shortest path between O and F in the corresponding graph.

**Example 22** Consider again Figure 5.22, which shows the graph induced by the set of sequences \( \mathcal{R} = \{ \text{abdc}, \text{bca} \} \). For instance, (2, 1) is the final node for the subproblem \( \mathcal{R}' = \{ \text{ab}, \text{b} \} \) (the 2- and 1-prefixes of the original sequences, respectively). By adding an edge c to any common supersequence of \( \{ \text{ab}, \text{b} \} \) we obtain common supersequences for \( \{ \text{ab}, \text{bc} \} \).
and therefore we insert edge c between nodes (2, 1) and (2, 2). The shortest path from the initial node O=(0, 0) to the final node F=(4, 3) is \{(0, 0), (1, 1), (2, 1), (3, 1), (4, 2), (4, 3)\}, which corresponds to SCS(\mathcal{R}) = abdca.

We now review an algorithm to solve the SCS problem that was presented in [NO01]. This algorithm generally results in fast executions because it does not need to materialize the whole graph in advance.

5.2.3.2 An A*-based Approach to Solve SCS

Algorithm A* [Nil71] is a heuristic search procedure to efficiently find shortest paths in graphs that are inductively built (i.e., graphs in which we can generate the set of successors of any given node). The key idea in [NO01] is to apply A* to the SCS problem and materialize only a portion of the graph induced by the input set of sequences. We now review the A* algorithm and explain how it can be used to solve SCS [NO01].

A* searches the input graph outwards from the starting node O until it reaches the goal node F, expanding at each iteration the node that is most likely to be along the best path from O to F. For each node u, A* relies on the ability to estimate a lower bound of the length of the best path connecting O and F using u; we denote this value \(f(u)\). At each step we choose the most promising node among the already created nodes, i.e., the node with smallest \(f(u)\). Then, we expand the chosen node by dynamically generating all its successors in the graph.

Typically, the cost function \(f(u)\) is the sum of two terms, \(f(u) = g(u) + h(u)\), where \(g(u)\) is the length of the shortest path found so far from O to u, and \(h(u)\) is the expected remaining cost (heuristically determined) to get from u to F. If the heuristic function \(h(u)\) is always an underestimate of the actual length of the shortest path from u to F, then A* is guaranteed to find the optimal solution. However, if \(h(u)\) is too optimistic, A* expands too many nodes and may run out of resources before a solution is found. Therefore, it is critical to define \(h(u)\) as tightly as possible.
CHAPTER 5. BUILDING SITS

An important property that results in efficient implementations of A* is as follows. Consider any pair of nodes \( u \) and \( v \) in the graph, and let \( d(u, v) \) be the cost of going from \( u \) to \( v \). If \( h(u) - h(v) \leq d(u, v) \) holds for every choice of \( u \) and \( v \), then whenever a node \( u \) is expanded, a shortest path from \( O \) to \( u \) is already known.

For the SCS problem, an estimate on the length of the shortest path from \( u \) to \( F \), \( h(u) \), is equivalent to an estimate of the shortest common supersequence of the suffixes of the original sequences not yet processed in state \( u \). A good value for \( h(u) \) can then be calculated as follows. If we denote as \( o(u, c) \) the maximum number of occurrences of \( c \) in some suffix sequence in state \( u \), a lower bound \( h(u) \) is then \( \sum_c o(u, c) \), since every common supersequence must contain at least \( o(u, c) \) occurrences of \( c \). For instance, consider the node \((2, 1)\) in Figure 5.22, for which we processed the two first elements of \( abdc \) and the first element of \( bca \). The remaining suffixes are \( dc \) and \( ca \), respectively. In this case, \( h((dc, ca)) = o((dc, ca), d) + o((dc, ca), c) + o((dc, ca), a) = 1 + 1 + 1 = 3 \).

<table>
<thead>
<tr>
<th>Algorithm A* for SCS (graph in implicit form; ( O, F ): nodes in the graph)</th>
</tr>
</thead>
<tbody>
<tr>
<td>01 OPEN = {O}; CLOSED={}; g(O)=0; f(O)=0</td>
</tr>
<tr>
<td>02 repeat</td>
</tr>
<tr>
<td>03 bestN = n ∈ OPEN such that f(n) is minimal</td>
</tr>
<tr>
<td>04 OPEN = OPEN - {bestN}</td>
</tr>
<tr>
<td>05 CLOSE = CLOSE ∪ {bestN}</td>
</tr>
<tr>
<td>06 gNew = g(bestN) + 1 // ( d(bestN, s) = 1 )</td>
</tr>
<tr>
<td>07 for each successor ( s ) of bestN do</td>
</tr>
<tr>
<td>08 if (s \notin OPEN ∪ CLOSE) ∨ (s ∈ OPEN ∧ gNew &lt; g(s)))</td>
</tr>
<tr>
<td>09 g(s) = gNew; h(s)=( \sum_c o(u, c) ); f(s)=g(s)+h(s)</td>
</tr>
<tr>
<td>10 OPEN=OPEN ∪ {s}</td>
</tr>
<tr>
<td>11 until (bestNode = ( F ))</td>
</tr>
</tbody>
</table>

Figure 5.23: Solving the shortest common supersequence problem using A*.

Figure 5.23 shows the A* algorithm to solve SCS. We note that, in principle, A* does not reduce the size of the graph. However, A* usually finds SCS values efficiently because A* does not generate the whole graph in advance, but rather is guided by the heuristic function \( h \) to explore a small fraction of the graph.
CHAPTER 5. BUILDING SITS

5.2.3.3 Exploiting SCS to Create Multiple SITs

We now describe how to map the problem of optimally creating a set of SITs into a generalized version of SCS. We then adapt the A* technique above to solve this problem and present greedy heuristics to approximate good solutions in a small amount of time.

We recall from the description of Sweep (see Section 5.2.2) that to create a SIT we perform some sequential scans over the tables referenced in the SIT’s generating query. Moreover, the sequential scans follow some post-order traversal of the join tree (see Section 5.2.2.3). As an example, suppose we want to create SIT\((R:a\j R \bowtie S \bowtie T \bowtie U \bowtie V)\) shown on the right of Figure 5.24. Clearly, sequential scans over tables \(S\) and \(U\) (which return SIT\((S:s_1\j S \bowtie t_1\) \(T\) and SIT\((U:u_1\j U \bowtie u_2=v_1\) \(V\)) must precede the sequential scan over table \(R\), since the latter uses SITs produced by the former. However, no ordering is required between sequential scans over \(S\) and \(U\).

We can concisely specify these order restrictions using dependency sequences. A dependency sequence is an ordered list of tables \((R_1,\ldots,R_n)\), such that for all \(1 \leq i < j \leq n\), the sequential scan of table \(R_i\) must precede the sequential scan of \(R_j\). For linear-join queries, a single dependency sequence is needed, which is obtained by traversing the sequence of joins. In general, for an acyclic-join query we need one dependency sequence for each root-to-leaf path in the join tree (omitting leaf nodes). Figure 5.24 shows two join queries and their corresponding dependency sequences.

Figure 5.24: Dependency sequences for join generating queries.
To model the time and space required to execute Sweep over a two-way join generating query, we associate each table $T$ with the following values: \( \text{Cost}(T) \), which is the estimated cost to perform a sequential scan over $T$, and \( \text{SampleSize}(T, a) \), which specifies how much memory we allocate for a sample over attribute $a$ of $T$. \( \text{SampleSize}(T, a) \) can be a constant value or depend on the specific table and attribute. Therefore, if we use Sweep to create \( \text{SIT}(S.a|R \bowtie S) \), \( \text{Cost}(S) \) refers to the cost of a sequential scan over $S$, and \( \text{SampleSize}(S, a) \) specifies the memory requirements for the samples of attribute $S.a$.

As illustrated in Example 20, we can share a sequential scan over table $S$ to create any SIT of the form \( \text{SIT}(S.a|R \bowtie_{x=y} S) \) for arbitrary table $R$ and attributes $a$, $x$ and $y$, provided that we have histograms $H(R . x)$ and $H(S . y)$ available (recall that for acyclic-join generating queries, $R$ could represent an intermediate join result). In this situation, the cost of executing Sweep remains fixed at \( \text{Cost}(T) \) since the sequential scan is shared. However, the space required for sampling increases to \( \sum \text{SampleSize}(T, a_i) \) for attributes $a_i$ that require samples while scanning $S$. For instance, if we share the sequential scan over $S$ to create \( \text{SIT}(S.a|R \bowtie_{x=y} S) \), \( \text{SIT}(S.b|R \bowtie_{x=y} S) \), and \( \text{SIT}(S.a|T \bowtie_{z=y} S) \), the estimated cost is \( \text{Cost}(S) \) and the memory requirements for sampling are \( 2 \cdot \text{SampleSize}(S, a) + \text{SampleSize}(S, b) \).

**Unbounded Memory**

If the amount of available memory is unbounded, the optimization problem can be very easily mapped to a weighted version of SCS, where the input sequences to the SCS problem are all the dependency sequences of the given SITs. All we need to change in the A* algorithm is the definition of the distance function between nodes to incorporate weights, and the heuristic function $h(u)$ (lines 6 and 9 in the A* algorithm of Figure 5.23).

In particular, we change $d(\text{bestN}, s)$’s value in line 6 from the constant 1 (no weight) to \( \text{Cost}(R) \), where $R$ is the label of edge (\( \text{bestN}, s \)). The definition of $h(u)$ is changed

---

\[3\text{We would need slightly more space than } \text{SampleSize}(T, a_i) \text{ to store the input histograms, but we omit this detail for simplicity.}\]
accordingly and the second assignment of line 9 becomes \( h(s) = \sum_c \text{Cost}(c) \cdot o(u,c) \).

Once we get the SCS that corresponds to a schedule of applications of \text{Sweep}, we “execute” it by iterating through the elements (tables) of the SCS one at a time. When we process table \( T \), we create (using \text{Sweep}) all SITs of the form \( \text{SIT}(T.a|S \gg_{s_i=t_j} T) \) for which the histogram \( H(S,s_i) \) is already built (or, if \( S \) is a base table, we create the corresponding base-table histogram).

**Example 23** Consider the SITs of Figure 5.24, and assume that \( \text{Cost}(R) = \text{Cost}(S) = 10 \), \( \text{Cost}(T) = \text{Cost}(U) = \text{Cost}(V) = 20 \), and \( \text{SampleSize}(t,a) = 10,000 \) for all tables and attributes. In such a case, a shortest weighted common supersequence with cost 60 is \((U, T, S, R)\). The execution of schedule \((U, T, S, R)\) proceeds as follows. We first perform a sequential scan over table \( U \) obtaining \( \text{SIT}(U.u_1|U \gg_{u_2=v_1} V) \). Then, with a sequential scan over \( T \) we obtain \( \text{SIT}(T.t_1|T \gg_{t_2=v_1} V) \). Next, we perform a sequential scan over \( S \) obtaining \( \text{SIT}(S.s_1|S \gg_{s_2=t_1} T) \) and \( \text{SIT}(S.s_1|S \gg_{s_2=t_1} T \gg_{t_2=v_1} V) \) (using \( 2 \cdot \text{SampleSize}(S,s_1) \) memory for samples). Finally, we do a sequential scan over table \( R \) and obtain, using \( \text{SampleSize}(R,a) + \text{SampleSize}(R,b) \) for samples, the required \( \text{SIT}(R.b|R \gg_{r_1=s_1} S \gg_{s_2=t_1} T \gg_{t_2=v_1} V) \) and \( \text{SIT}(R.a|R \gg_{r_1=s_1} S \gg_{s_2=t_1} T \gg_{t_2=u_1} U \gg_{u_2=v_1} V) \).

The scenario considered above assumes that we can allocate any amount of memory to create SITs. When the amount of available memory \( M \) is bounded, we need to additionally modify the search space to solve a constrained, weighted SCS. For instance, if \( 2 \cdot \text{SampleSize}(S,s_1) > M \) in the example above, we would not be able to share the sequential scan over \( S \), and the optimal execution path would be different. We now show how to modify A* to obtain the optimal strategy to create multiple SITs given memory constraints.
CHAPTER 5. BUILDING SITS

\begin{center}
\begin{tabular}{|l|}
\hline
\textbf{generateSuccessors} \((u=(u_1,\ldots,u_n)\):node, \(\mathcal{R}=[R_1,\ldots,R_m]\):sequences, \(M\):memory) \\
1 \hspace{1em} succ = \emptyset \\
2 \hspace{1em} for each table \(T\) in \(\mathcal{R}\) do \\
3 \hspace{2em} cand = \{i : R_i[u_i] = T\} \\
4 \hspace{2em} for each \(C \subseteq\) cand such that \(\sum_{a_i} \text{sampleSets}(a_i) \cdot \text{SampleSize}(T,a_i) \leq M\) do \\
5 \hspace{3em} succ = succ \cup \{(v_1,\ldots,v_n)\}, \text{ where } v_i = \begin{cases} 
    u_i + 1 & \text{if } i \in C \\
    u_i & \text{otherwise}
\end{cases} \\
6 \hspace{1em} return succ \\
\hline
\end{tabular}
\end{center}

Figure 5.25: Obtaining successor nodes in the modified A* algorithm.

Bounded Memory

The main implication of having a bounded amount of available memory is that some edges in the A* search graph are no longer valid. As explained in Section 5.2.3.2, the implicit meaning of an edge from node \(u = (u_1,\ldots,u_n)\) to node \(v = (v_1,\ldots,v_n)\) with label \(\omega\) is to “advance” one position all input sequences for which \(R[v_i] = \omega\). Of course, while creating SITs, each position that was changed from \(u_i\) to \(v_i = u_i + 1\) in transition \((u,v)\) corresponds to an additional SIT to create and might increase the memory requirements beyond the limit. For that reason, in general we can only advance \textit{subsets} of all possible positions from node \(u\) using edge \(c\). We modified the procedure to get the successors of a given node at each iteration of A* to solve this constrained version of SCS. The modified procedure is shown in Figure 5.25.

We note that the size of the search graph is still bounded by \(O(l^n)\), where \(n\) is the number of input SITs and \(l\) is the size of the largest dependency sequence among the input SITs. However, the number of edges typically increases.

Alternative Heuristics

The A* algorithm is guaranteed to find an optimal schedule to create a set of SITs that minimizes the total execution cost. However, the worst-case time complexity of the algorithm is \(O(l^n \cdot 2^S)\) where \(l\) is the maximum length of any chain of joins, \(n\) is roughly the number of input SITs, and \(S\) is the maximum size of any candidate set. For small values
of $l$ and $n$, the $A^*$ algorithm is efficient, but if we increase $n$ or $l$ some executions of $A^*$ become prohibitively expensive. For that reason, we now propose a simple greedy variant of the $A^*$ algorithm and also a hybrid approach that balances efficiency and quality of the resulting schedule.

- The *Greedy* approach can be described as a simple modification of $A^*$. At each iteration of $A^*$, after we get the best node $u$, we empty set $\text{OPEN}$ before adding the successors of $u$. In this way, the *Greedy* variant simply chooses at each step the element that would result in the largest local improvement. In this case, the size of $\text{OPEN}$ at each iteration is bounded by the maximum number of successors of any given node, and the algorithm is guaranteed to finish in at most $\sum_i |R_i|$ steps (since the induced search graph is always acyclic). However, due to the aggressive pruning in the search space, *Greedy* usually results in suboptimal schedules.

- *Hybrid* is a combination of *Greedy* and $A^*$ that is based on the observation that we can switch from $A^*$ to *Greedy* at any given iteration by simply cleaning $\text{OPEN}$ at the current and every subsequent iteration. *Hybrid* starts as $A^*$ and after a switching condition becomes true, *Hybrid* continues in a greedy way from the most promising node found so far. Several switching conditions can be used for *Hybrid*. For instance, we can switch after a pre-determined amount of time has passed without $A^*$ returning the optimal solution, or after $\text{OPEN} \cup \text{CLOSE}$ uses a predetermined amount of memory. In our experiments, we switch after one second without $A^*$ finding the optimal solution.

### 5.2.4 Experimental Evaluation

In this section we present experimental results for a C++ implementation of the algorithms described in Sections 5.2.2 and 5.2.3. In Section 5.2.4.1 we compare the proposed variants of *Sweep* to create a single SIT in terms of their accuracy. Then, in Section 5.2.4.2 we evaluate the techniques that simultaneously create multiple SITs.
5.2.4.1 Creating a Single SIT

We compare different techniques to build a single SIT using the following setting:

**Data Sets, Generating Queries, and Histograms:** We used a synthetic database consisting of 4 tables with 10,000 to 100,000 tuples. Each table has three to five attributes. Some attributes are uniformly distributed and others follow a Zipfian distribution with parameter $z$ varying from 0.1 to 1. The SIT generating queries are two-, three-, and four-way join queries. We used a variant of MaxDiff histograms [PIHS96] as the internal representation of SITs.

**Techniques Compared:** We compare *Sweep* (with a sampling rate of 10%) and its variations of Section 5.2.2.2. We also included *Hist-SIT*, the traditional technique used by current optimizers to estimate SITs by propagating base-table histograms. *Hist-SIT* is very efficient, since it operates on the histogram domain without accessing the actual data, but relies on all the simplifying assumptions of Section 3.1.

**Evaluation Metric:** For each technique, we created the corresponding SIT and also materialized the generating query to obtain the actual result. Then, we issued 1,000 random range queries over the SIT domain (corresponding to SPJ queries over the actual data) and calculated the relative error between the actual and estimated cardinalities.

**Results:** Figure 5.26 shows the relative error of the different techniques for generating queries of increasing complexity. As expected, *Hist-SIT* is consistently much worse than the other techniques, due to all the simplifying assumptions involved in its estimations. Moreover, the gap between *Hist-SIT* and the other techniques increases with the number of joins in the generating query, due to the propagation of errors through the query plans. For instance, in Figure 5.26(c) the relative error for *Hist-SIT* is almost 200% for all histogram sizes that we tried. In contrast, the variants of *Sweep* result in estimation errors that are close to those for the most accurate *SweepExact*. In particular, *Sweep*
Figure 5.26: Creating SITs with skewed join generating queries.
results in slightly worse estimations than *SweepFull* and *SweepIndex*, since it relies on additional simplifying assumptions (the difference, however, is small). Also, there is no clear winner between *SweepFull* and *SweepIndex* in all situations. We also conducted an experiment in which the join predicates were uniformly distributed and independent of the remaining attributes. In this case, the independence assumption holds, and all techniques result in very accurate estimations (with relative errors below 2%).

### 5.2.4.2 Creating Multiple SITs

We now study our proposed techniques to create multiple SITs, using the following setting.

**Data Sets and Generating Queries:** We generated data schemas and SITs using several parameters. In particular, we first created *nt* different table schemas. The number of tuples in each table follows a Zipfian distribution with parameter \( z = 1 \). We set \( \text{Cost}(T) = |T|/1000 \) units (since the cost of a sequential scan is proportional to the size of the input), and \( \text{SampleSize}(T) = s \cdot |T| \), where \( s \) is the sampling rate. Finally, we randomly generating *numSits* dependency sequences, each one with a length between two and *lenSits*. By default we used the values *numSits*=10, *lenSits*=5, *nt* = 10 and *s* = 10%. The combined size of all tables was set to 1,000,000.

**Techniques Compared:** We compare the following techniques: *Naive* (which creates each SIT separately), *Opt* (the optimal strategy of Section 5.2.3.3 based on A*), *Greedy*, and *Hybrid*. By default, we specified the available amount of memory as \( M = 50,000 \).

**Evaluation Metric:** For each experiment, we generated 100 different instances (sets of SITs) and optimized their evaluation strategies using each technique. We finally averaged, for each technique, the total optimization time used to find each schedule, and the estimated cost of such schedule.

\[4\text{We compare different algorithms that return schedules to create SITs, but we do not actually execute those schedules. Therefore, we do not populate the tables, but only specify their schemas and sizes.}\]
Varying numSits and lenSits: Figure 5.27 shows the estimated cost and optimization time for the default setting and varying numSits. Not surprisingly, Naive results in considerably slower schedules than the remaining techniques, since it does not take advantage of commonalities between SITs. Both Greedy and Hybrid result in close-to-optimal schedules for all scenarios (see Figure 5.27(a)). However, the optimization times of Greedy and Hybrid are substantially smaller than those of Opt, especially for larger instances. For instance, for sets with 20 SITs, Opt took over 36 seconds on average to find the optimal solution. In contrast, Greedy and Hybrid took 0.02 and 0.8 seconds, respectively, to produce slightly less efficient schedules (the average estimated costs were 2024, 2124, and 2278 units for Opt, Greedy, and Hybrid respectively). We also conducted experiments varying lenSits and obtained similar results (see Figure 5.28).
CHAPTER 5. BUILDING SITS

Figure 5.28: Creating SITs with varying lenSits.

Varying the Total Number of Tables: Figure 5.29 shows the average estimated cost to create SITs for varying number of tables. When we increase nt (keeping numSits fixed) the amount of overlap among SITs is reduced. For that reason, all techniques tend to return similar schedules. In the extreme case in which every SIT consists of different tables, the Naive approach is optimal, since it performs just one sequential scan over each involved table. In real applications, however, the degree of commonality among SITs usually is fairly high and thus the use of alternatives to Naive becomes important.
CHAPTER 5. BUILDING SITS

Figure 5.29: Creating SITs with varying number of tables.

Varying the Amount of Available Memory $M$: In this last experiment, we varied the amount of available memory $M$ from 15 KB, which is the sample size of the largest table and therefore the minimal amount of memory for any algorithm, to 45 KB, which resulted in the same schedule as when $M$ is not bounded (see Figure 5.30). *Naive* is not affected by memory constraints, since it maintains a single sample in memory at all times. The remaining techniques clearly benefit from the extra memory and their schedules are executed in almost half of the time required by *Naive*.

Figure 5.30: Creating SITs with varying memory limit.
5.3 Conclusion

In this chapter we presented several techniques to efficiently create statistics on query expressions. The main contributions of this chapter are summarized below:

- We proposed a feedback-driven architecture to build and refine SITs in a database system. Specifically, we focused on \textit{STHoles} histograms as the structure for SITs and showed that, after a few iterations, we can obtain accurate statistical estimators imposing low overhead to the query processor.

- We introduced a family of techniques to build SITs off-line whenever the feedback-driven procedure above is not adequate. In particular,
  - We introduced \textit{Sweep}, a family of techniques to efficiently create SITs that avoids relying on the independence assumption between attributes and requires a single scan over the tables involved in the SIT’s generating query. The basic formulation of \textit{Sweep} uses sampling and relies on the containment assumption for efficiency. We showed experimentally that these assumptions do not significantly reduce the accuracy of the resulting SITs.
  - We studied the problem of scheduling basic applications of \textit{Sweep} to create a set of SITs subject to space constraints. We identify this problem as a generalization of the well-known “Shortest Common Supersequence” problem, and proposed both exact and approximate algorithms to obtain good schedules. We showed experimentally that our heuristic approaches obtain a close-to-optimal sequence of applications of \textit{Sweep} in a small fraction of the time taken by the optimal technique.
Chapter 6

Recommending SITs

The full set of base-table statistics that are useful to optimize a query is usually small and can be statically inferred from the query using simple syntactic analysis. For unidimensional statistics, this set has at most one histogram per attribute that is mentioned in the input query (and excludes those attributes that are only mentioned in the query’s SELECT clause). For that reason, the selection and creation of base-table statistics in current optimizers can be fully automated as follows. Whenever a required histogram is not available during the optimization of an input query, the optimizer suspends optimization, efficiently materializes the missing histogram by leveraging indexes or using sampling over base tables, and finally resumes optimization. This procedure is a very effective tool to automatically decide which base-table statistics to build – and when to build them – in a modern RDBMS.

When we add SITs into the picture, the situation becomes more complex. In general, any query sub-plan that is explored by the optimizer might exploit different candidate SITs. Besides, unlike the case for base-table statistics, creating SITs incurs some processing overhead that is not trivial during optimization. For instance, the efficient techniques of Section 5.2 perform a number of sequential scans over the underlying data. This observation rules out generalizing the approach taken by current optimizers regarding base-table statistics to SITs: creating all potentially useful SITs on demand significantly
hurts the performance of the optimizer. Furthermore, loading, maintaining, and exploiting SITs for all possible intermediate results is not viable even for simple schemas.

In Chapter 5 we introduced an online feedback-based mechanism to build and refine SITs as queries are answered by the RDBMS. We showed that we can build accurate SITs without evaluating their generating queries, and that the choice of SITs to refine can be guided by the specific set of queries executed in the RDBMS. We also discussed why this approach is not appropriate in all scenarios. One reason is that SITs that might be useful during optimization cannot always be built by just exploiting feedback from user queries. Also, the above approach is designed for statistics that can be built and refined using query feedback (such as, for instance, $STHoles$ histograms). If the RDBMS does not support $STHoles$ or other appropriate statistics natively, the feedback-based approach of Chapter 5 cannot be applied. Finally, in Section 5.1 we proposed simple heuristics to choose which SITs to refine for a query that is executed. However, these simple approaches use local information to guide the selection process (taken from the query being currently executed) and tend to generate a large number of statistics over time. Moreover, it is not clear how to pick a small but useful subset of those SITs if memory is a constraint.

For those reasons, in this chapter we explore an alternative approach to recommend a small number of SITs to materialize. We use the framework of Section 4.2, based on conditional selectivity values, as the engine to exploit SITs during cardinality estimation. Specifically, in this chapter we focus on recommending unidimensional SITs with SPJ generating queries. Our approach is inspired by work on automatic index selection [CN97], which takes a query workload as input and recommends which indexes to materialize in the database system so that subsequent queries coming from similar workloads are executed more efficiently. Analogously, the techniques in this chapter guide the selection of SITs using workload information and query feedback. In contrast to the technique of Section 5.1, the new approach is not restricted to a specific family of histograms and works off-line, therefore taking into account global information about workload characteristics.
The rest of the chapter is structured as follows. In Section 6.1 we formally define the problem of recommending SITs to materialize in an RDBMS. Then, in Section 6.2 we introduce a workload-based technique to choose a small but useful set of SITs to materialize. Finally, in Section 6.3 we report an experimental evaluation of our technique over a commercial RDBMS.

### 6.1 Problem Formulation

The ultimate goal of our techniques is to find a small number of SITs to materialize in an RDBMS, so that the optimizer substantially increases the quality of subsequent execution plans by exploiting such SITs during cardinality estimation. In this section, we state some simplifying assumptions that make the problem more tractable, and then precisely specify the problem that we address in Section 6.2.

#### 6.1.1 From Execution Cost to Cardinality Estimation

As discussed in Chapter 2, the optimizer relies on a cost model to choose the most efficient execution plan for an input query. This cost model is heavily based on cardinality estimations of sub-plans generated during optimization, but also includes other components. This is illustrated in the following example.

**Example 24** Consider the following query:

```sql
SELECT * FROM R WHERE R.a < 10
```

One possible execution plan for the query performs a sequential scan over table R and filters on the fly all tuples that satisfy \( R.a < 10 \). Alternatively, if an index over \( R.a \) is available, another possibility is to obtain the record identifiers of all tuples in R that satisfy \( R.a < 10 \) using the index, and then perform record lookups to get the actual columns that belong to the result. To choose between these two alternatives, the optimizer first estimates the number of tuples that satisfy \( R.a < 10 \) and then uses that information
to model the execution cost of each plan. For the first plan, the optimizer estimates how expensive it is to sequentially read all pages from disk with table $R$ tuples. For the second plan, the optimizer estimates how much time it takes to fetch from disk – using random accesses – all pages that might contain relevant record identifiers (the number of such pages is determined using the estimated number of tuples satisfying $R.a < 10$). Finally, the optimizer selects the alternative with the smallest expected execution cost.

We note that to model the execution time of different alternative plans, the optimizer approximates the cost of various operations on disk (e.g., the cost of reading pages sequentially from disk, or the cost of randomly fetching a page from disk). For more complex operators, the optimizer might additionally estimate the amount of available memory during the evaluation of each plan.

For that reason, even if we use exact cardinality values, the estimated cost of the resulting query plans might still be slightly inaccurate. However, it is observed that in practice cardinality estimation is the main source of problems during optimization. If cardinality estimations are accurate, the estimated cost of the resulting plans might be generally off by at most 10 percent. Thus, a simplifying assumption that we use in this chapter is that the only source of errors during optimization results from inaccuracies in cardinality estimates. Therefore, in order to obtain efficient execution plans we need to reduce as much as possible errors during cardinality estimation. We then relax our problem statement to finding a small set of SITs to materialize in an RDBMS so that cardinality estimates for incoming query sub-plans are as accurate as possible.

### 6.1.2 Using Workload Information

Similar to work done in the context of automatic selection of indexes and materialized views (e.g., [CN97, ACN00]), we use workload information to guide the choice of SITs to materialize. An important difference between the work in [CN97, ACN00] and our setting is that techniques that recommend indexes or materialized views can accurately model the impact of, say, a candidate index using hypothetical configurations. This is possible
because the optimizer only uses information about the availability of a candidate index or materialized view to estimate the impact of such structures during execution. In contrast, there is not an easy way to evaluate the impact of a given SIT without building such SIT in advance. (As mentioned earlier, there is a very large number of candidate SITs to consider, and materializing each one of them in advance is not viable.) To mitigate this problem, in our algorithms we use additional information about the workload to identify useful SITs. We assume that the queries in the input workload were previously executed, and additional information was extracted from such executions. Specifically, when evaluating query $q$, we record the execution plan that was chosen to evaluate $q$, the time it took to evaluate such execution plan, and cardinality information on different operators in the execution plan. We use actual execution times to measure the importance of a query, and cardinality feedback on plan operators to guide the selection of SITs. Therefore, each query $q$ in the workload is associated with the following information:

$$ \left[ \text{execTime}_q, \{(\text{sp}_1^q, \text{est}_1^q, \text{act}_1^q), \ldots, (\text{sp}_n^q, \text{est}_n^q, \text{act}_n^q)\} \right] $$

where $\text{execTime}_q$ is the actual execution time of $q$, and each $(\text{sp}_i^q, \text{est}_i^q, \text{act}_i^q)$ provides additional information about the operators in the actual query plan that was executed. Specifically, if the executed plan for $q$ consists of $n$ operators, $\text{sp}_i^q$ represents the $i$-th operator in the plan (using some fixed arbitrary order), $\text{est}_i^q$ is the estimated cardinality of $\text{sp}_i^q$ using the set of statistics that was available when the query was executed, and $\text{act}_i^q$ corresponds to the actual number of tuples produced by $\text{sp}_i^q$. We note that $\text{act}_i^q$ values can be obtained from the execution of $q$ with virtually no overhead: unlike the procedure of Section 5.1 that identifies the specific bucket containing each streamed tuple, in this case we just need to record the total number of tuples produced at $\text{sp}_i^q$.

As an example, consider the execution plan in Figure 6.1(b), which was used to evaluate the query in Figure 6.1(a). Figure 6.1(c) shows the information that is gathered for such query during the execution of the corresponding query plan. The cardinality of $\sigma_{S,a<10}(S)$ is accurately estimated (57 estimated tuples versus 59 actual tuples), and so it is the cardinality of $\sigma_{T,b>20}(T)$. In contrast, the cardinality of $\sigma_{S,a<10}(R \bowtie S)$ and
the whole query $\sigma_{S,a<10 \land T,b>20}(R \bowtie S \bowtie T)$ are poorly estimated (in particular, the estimated cardinality of $\sigma_{S,a<10}(R \bowtie S)$ is around 3% of the actual value). If the input workload consists of just the query in Figure 6.1, the query optimizer would most likely benefit from SITs that accurately estimate the cardinality of $\sigma_{S,a<10}(R \bowtie S)$.

Using this extended workload information, we reformulate the problem of recommending SITs to materialize in an RDBMS as follows:

Let $k$ be an integer and $W = \{q_1, \ldots, q_i\}$ a query workload, where each $q_i$ is an SPJ query associated with values $execTime_q$ and

\[
\{(sp_q^1, est_q^1, act_q^1), \ldots, (sp_q^n, est_q^n, act_q^n)\}.\]

We want to find the set of $k$ unidimensional SITs that are expected to minimize the estimation error $|est_q^i - act_q^i|$ of each sub-plan $sp_q^i$ in $W$.

In the formulation above, we implicitly assume that by improving cardinality estimates of all sub-plans $sp_q^i$, we would also indirectly benefit the remaining sub-plans that were explored but not chosen for execution during optimization of $q$. While this assumption clearly is a simplification, we found experimentally that the recommended SITs are also useful to estimate the cardinality of sub-plans beyond those previously executed. (We report experimental results in Section 6.3.)
6.1.3 Challenges Associated with SIT Selection

We conclude this section by summarizing the main obstacles to effectively recommend SITs in an RDBMS:

- A recommendation of SITs is as good as the optimizer that uses those SITs. That is, suppose that we know that a particular SIT can dramatically improve the cardinality estimation of a query. If for some reason the optimizer ignores this SIT during cardinality estimation, then the presence of such SIT in the RDBMS is not beneficial.

- For efficiency purposes, the recommendation of SITs must be obtained without materializing any candidate SIT in advance. As argued above, the impact of a candidate SIT that is not materialized is more difficult to estimate than that of a candidate index that has not yet been created.

- During optimization, the impact of one SIT might change substantially depending on the actual available SITs. For instance, $S_1 = \text{SIT}(S.a | R.s = S.s, S.t = T.t)$ might be useful for optimization in the absence of other SITs, but it can have only limited impact if $\text{SIT}(S.a | R.s = S.s)$ is also available. Dependencies between SITs can be complex when considering decompositions. In fact, the presence of a new SIT can completely change the most accurate decomposition of a query, therefore altering the relative impact of multiple other SITs that in principle are unrelated to the new one.

6.2 A Workload-based Approach to Recommend SITs

In this section we introduce our approach to recommend a small but useful set of SITs for an input query workload, which makes the optimizer choose better quality execution plans for subsequent queries. Figure 6.2 shows an overview of the architecture of the proposed technique. The dotted line in the figure delimits the boundary between the
RDBMS and the recommendation module, which is implemented as a client. For a given workload, we first generate the set of candidate SITs that might reduce cardinality errors by syntactically analyzing the queries (Section 6.2.1). Then, the what-if analysis component (Section 6.2.2) is repeatedly used to approximate the resulting error in cardinality estimation when given candidate SITs (which have not been necessarily materialized) are available. Using this component, a selection module (Section 6.2.3) chooses the candidate SITs to materialize, which are expected to reduce cardinality estimation errors for subsequent queries. We now analyze each component in detail.

### 6.2.1 Candidate Generation

Initially, our proposed technique generates all syntactically valid SITs that could be exploited to estimate the cardinality of any sub-plan $sp^i_q$ in the workload. We emphasize that this step only generates definitions of relevant SITs, and does not materialize any SIT in advance.
In general, each sub-plan $sp^i_q$ in the workload is an SPJ query, and can therefore be regarded as a set of both filter and join predicates applied over a cartesian product of relations. Then, for a given $sp^i_q$ associated with predicates $P = \{p_1, \ldots, p_n\}$, we generate all unidimensional SITs that can be used to estimate $Sel_R(P)$. Recall from Chapter 4 that by applying atomic decompositions, $Sel_R(P)$ can be expressed as $Sel_R(p_i|P - p_i)$ for each $p_i \in P$. Therefore, the set of candidate SITs to approximate $Sel_R(P)$ is the union, over all $p_i$, of the candidate SITs that can be used to approximate either $Sel_R(p_i|P - p_i)$ or (recursively) $Sel_R(P - p_i)$. The candidate SITs for $Sel_R(p_i|P - p_i)$ depend on the actual type of predicate $p_i$:

**Case 1:** $p_i$ is a filter predicate over attribute $a$ (e.g., $a < 10$). In this case, according to Section 4.2.2.3, the set of candidate SITs to approximate $Sel_R(p_i|P - p_i)$ is:

$$\{\text{SIT}(a|Q) \text{ such that } Q \subseteq P - \{p_i\}\}.$$

**Case 2:** $p_i$ is a join predicate (e.g., $R.s=S.s$). In this case, the set of candidate SITs to approximate $Sel_R(p_i|P - p_i)$ can be obtained by using a similar procedure to the one discussed in Section 4.2.2.3. First, we transform $p_i$ into a pair of wildcard selection predicates (in the example, $R.s=?$ and $S.s=?$). Then, we apply the separable decomposition property to $Sel_R(P - R.s=S.s, R.s= ?, S.s= ?)$ obtaining $Sel_R(R.s= ?, P_1) \cdot Sel_R(S.s= ?, P_2)$. Finally, we obtain the candidate SITs for both $Sel_R(R.s= ?|P_1)$ and $Sel_R(S.s= ?|P_2)$, as explained for the previous case.

To efficiently generate all candidate SITs for a query workload, we process each query independently and progressively update a global table that contains the relevant SITs. To identify the candidate SITs for a query $q$ we use the following procedure, which is illustrated in Figure 6.3. We first obtain the node $sp^i_q$ that corresponds to the full execution plan for $q$ (see Figure 6.3(a)). Suppose that $sp^i_q$ is associated with predicates $P = \{p_1, \ldots, p_n\}$. Any sub-plan $sp^j_q$ of $sp^i_q$ must be associated with predicates $P' \subset P$. Therefore, the set of candidate SITs for $sp^j_q$ is included in that of $sp^i_q$. In fact, after some applications of the atomic decomposition property, $Sel_R(P')$ is just a factor of $Sel_R(P)$. 
(a) Query execution plan. (b) Lattice of predicates. (c) SITs generated for a given lattice node.

Figure 6.3: Generating candidate SITs for a query.

Therefore, we next generate the lattice of predicates associated with \( sp_i^q \), as partially shown in Figure 6.3(b). It is fairly easy to show that we can generate all candidate SITs for \( sp_i^q \) (and therefore for \( q \)) by traversing the lattice of predicates and processing each node that corresponds to a non-separable decomposition (identified using thick ovals in the figure) as explained next. Consider a node in the lattice associated with predicates \( P = \{ p_1, \ldots, p_n \} \). We process each predicate \( p_i \) in turn, generating one (or two) SITs for \( p_i \) as follows (to simplify the presentation, we define \( Q_i = P - \{ p_i \} \)):
**Case 1:** If \( p_i \) is a filter predicate (e.g., \( R.a < 10 \)), we generate \( \text{SIT}(R.a|Q_i) \) (the remaining \( \text{SIT}(R.a|Q'_i) \) with \( Q'_i \subseteq Q_i \) are generated at other nodes in the lattice).

**Case 2:** If \( p_i \) is a join predicate (e.g., \( R.s=S.s \)), we separate \( \text{Sel}_R(Q_i; R.s=?; S.s=?) \) into \( \text{Sel}_R(R.s=?; Q'_1) \cdot \text{Sel}_R(S.s=?; Q'_2) \) and then generate candidates \( \text{SIT}(R.s|Q'_1) \) and \( \text{SIT}(S.s|Q'_2) \) (the remaining SITs over attributes \( R.s \) and \( S.s \) are generated at other nodes in the lattice).

The above procedure is shown in Figure 6.3(c) for the sub-plan associated with predicates \( \{R.t=T.t; S.a < 10, T.b > 5, R.s=S.s\} \). For instance, for \( p_i = S.a < 10 \) we just generate the candidate \( \text{SIT}(S.a|R.t=T.t; T.b > 5, R.s=S.s) \). As another example, for \( p_i = (R.t=T.t) \) we first separate \( \text{Sel}_R(S.a < 10; T.b > 5, R.s=S.s, R.t=?, T.t=?) \) as \( \text{Sel}_R(R.t=?, R.s=S.s, S.a < 10) \cdot \text{Sel}_R(T.t=?, T.b > 5) \) and then generate candidates \( \text{SIT}(R.t|R.s=S.s, S.a < 10) \) and \( \text{SIT}(T.t|T.b > 5) \).

We summarize the algorithm to generate the set of candidate SITs for a query workload in Figure 6.4.

```
**generate_candidate_SITs** (W: workload)
returns (set of candidate SITs for W)
01 Initialize global table of candidates Cand = ∅
02 for each query q ∈ W
03 P = set of predicates in q
04 L = lattice of subsets of P
05 for each node l = \{p_1, ..., p_k\} ∈ L
06 if Sel(p_1, ..., p_k) is non-separable
07 for each predicate p_i
08 Generate SIT(s) to approximate Sel(p_i|P-p_i)  // see Section 6.2.1
09 Add generated SITs to Cand if not already present
10 return set of SITs in global table Cand
```

Figure 6.4: Generating candidate SITs for a workload.
6.2.2 What-if Analysis

After obtaining the set of candidate SITs to materialize for the input workload, we need to search this large space of alternatives and determine the best set of SITs, also called configuration, to recommend. A critical component in this process is the what-if module (see Figure 6.2). The what-if module essentially assigns a score to every possible configuration, and therefore provides a mechanism to rank alternative recommendations. In this section we discuss the what-if component in detail, and in the next section we explain how we use this component to recommend SITs.

Consider configuration $C$, which consists of a set of candidate SITs. The error of $C$ with respect to workload $W$ simply aggregates the expected inaccuracy in cardinality estimation of each query $q \in W$ weighted by $q$’s execution time. That is,

$$error(C, W) = \sum_{q \in W} execTime_q \cdot error(C, q)$$

The value $execTime_q$ is the actual execution time of $q$ when it was originally evaluated and added to workload $W$. Therefore, $error(C, W)$ gives more importance to queries that originally took longer to execute. In turn, we define the error of configuration $C$ with respect to query $q$ as the sum of expected errors in cardinality estimation over all sub-plans $sp^i_q$ in $q$’s execution plan. Specifically,

$$error(C, q) = \sum_{sp^i_q} |est^i_q - act^i_q| \cdot \frac{error(C, sp^i_q)}{error(C_W, sp^i_q)}$$

where $error(c, p)$ is the expected error—as returned by $getSelectivity$ in Section 4.2.2.2—of the most accurate decomposition to estimate the selectivity of plan $p$ using a given configuration $c$, and $C_W$ is the initial configuration available when $q$ was executed and added to $W$. The error of configuration $C$ with respect to query $q$ is then the sum, over each sub-plan $sp^i_q$ associated with $q$, of the expected error in estimation for $sp^i_q$ when $C$ is the set of available SITs. Specifically, we obtain the expected error in estimation for sub-plan $sp^i_q$ as follows. We first calculate the original estimation error for $sp^i_q$, that is, $|est^i_q - act^i_q|$. Then, we scale down this value by the expected decrease in error when
estimating the cardinality of \( sp_q^j \) using \( C \) instead of \( C_W \). That is, we multiply \( |est_q^j - act_q^j| \) by the fraction \( error(C, sp_q^j)/error(C_W, sp_q^j) \), effectively assuming that the decrease in error of the best decomposition is proportional to the decrease in error of the cardinality estimate produced by such decomposition.

In Chapter 4 we evaluated different error functions to compare the accuracy of alternative decompositions, and we showed that \( Diff \) consistently returns the best results (see Section 4.2.2.4 for the definition of \( Diff \) and Section 4.3.2 for experimental results). Using \( Diff \), the error of decomposition \( S = Sel_{R_1}(P_1|Q_1) \ldots Sel_{R_n}(P_n|Q_n) \) when approximated using \( H_1, \ldots, H_n \), respectively, is given by:\(^1\)

\[
Diff \left( \{\langle Sel_R(P_i|Q_i), H_i \rangle, i = 1 \ldots n \} \right) = \frac{\sum_i |Q_i| \cdot (1 - \text{diff}_{H_i})}{\sum_i |Q_i|}
\]

where \( \text{diff}_{H_i} \) measures the difference between the distribution of tuples modelled by SIT \( H_i \) and the corresponding base-table histogram (e.g., between the distributions modelled by \( H_i = \text{SIT}(R.a|R \bowtie S) \) and \( H(R.a) \)).

For a given SIT \( H \), the value \( \text{diff}_{H} \) can be calculated by processing \( H \) along with the corresponding base-table histogram (see Section 4.2.2.4). Unfortunately, obtaining \( \text{diff} \) values for the candidate SITs in our scenario using that procedure would require materializing every candidate SIT, which is not practical. Instead, we estimate \( \text{diff} \) values for the candidate SITs that are not materialized, and use these estimates in the equations for \( error(C, sp_q^j) \). In the next section we discuss how we estimate \( \text{diff} \) values for candidate SITs that are not materialized.

### 6.2.2.1 Estimating \( \text{diff} \) Values

The value \( \text{diff}_{H} \) for a SIT \( H = \text{SIT}(S.a|Q) \) quantifies the difference of the distribution of attribute \( a \) over its base table \( S \) and that over query expression \( Q \). As explained in Section 4.2.2.4, we calculate this value by manipulating \( \text{SIT}(S.a|Q) \) and base-table histogram \( H(S.a) \). In this section we present a mechanism to estimate \( \text{diff} \) values when the actual

\(^1\)Since we focus on unidimensional SITs, \( |P_i| \) values in the definition of \( Diff \) are always equal to one.
CHAPTER 6. RECOMMENDING SITS

SIT(\(S.a|Q\)) is not materialized. In particular, we use existing base-table histograms to perform this approximation. We now discuss two extreme approaches to estimate selectivity values that are the basis of our technique to approximate \(\text{diff}\) values, which we introduce next.

**Extreme Cardinality Estimation**

As explained in Chapter 2, cardinality estimation routines assume independence between attributes to estimate the result size of complex query plans. We now review such procedure using a simple example, and next we show how to modify the basic approach using other (extreme) assumptions to estimate selectivity values. Consider the query:

```sql
SELECT * FROM R,S
WHERE R.s=S.s AND S.a<10
```

and suppose that we estimate the cardinality of the query result by first obtaining the cardinality of \(\sigma_{S.a<10}(S)\), then propagating histogram \(H(S.s)\) through the filter predicate, and finally estimating the cardinality of the join between \(H(R.s)\) and the propagated histogram \(H(S.s)\) (see Section 2.2.2.3 for more details). For concreteness, consider now histograms \(H(R.r)\) and \(H(S.s)\), which are shown in Figure 6.5. We represent the number of distinct values in each bucket by equidistant vertical lines between bucket boundaries, and we use \(\delta\) to denote the density of each bucket (the density of bucket \(b\) is defined as \(b\)'s frequency divided by the number of distinct values in \(b\)). For instance, there are three groups of 20 tuples each in the first bucket of histogram \(H(S.s)\), and two groups of 10 tuples each in the first bucket of histogram \(H(R.s)\). Therefore, \(S\) consists of \(3 \cdot 20 + 2 \cdot 100 + 2 \cdot 4 = 268\) tuples and \(R\) consists of \(2 \cdot 10 + 2 \cdot 5 + 3 \cdot 50 = 180\) tuples.

For each pair of buckets from \(H(R.s)\) and \(H(S.s)\), the top of Figure 6.5 shows the number of tuples of \(S\) that can be joined with tuples of \(R\) under the containment assumption\(^2\). For instance, the expression \([2 \cdot 20 \ S \cdot 10 R]\) above the first pair of buckets specifies that two groups of 20 tuples each in \(S\) match with 10 tuples in \(R\) each. In the

\(^2\)The values at the bottom of the figure convey the same information for \(R\).
same way, the expression $[1 \cdot 20 \cdot S \cdot 0 \cdot R]$ specifies that the remaining group of 20 tuples in $S$ has no matches in $R$. By adding all such expressions we obtain the value 1800, which is the estimated cardinality of the join (ignoring the filter predicate).

Now suppose that we previously estimated, using histogram $H(S.a)$, that only 30 of the 268 tuples in $S$ satisfy predicate $S.a < 10$. To estimate the cardinality of the query, we traditionally scale down $H(S.s)$’s bucket densities by the factor $30/268$ so that the total number of tuples in the propagated $H(S.s)$ is 30. By using this procedure, we assume that the 30 tuples satisfying $S.a < 10$ are uniformly distributed in $S.a$’s domain and therefore each bucket in the propagated $H(S.s)$ holds a number of tuples satisfying $S.a < 10$ that is proportional to its original frequency. It is easy to verify that when estimating the join of the original $H(R.s)$ and the propagated $H(S.s)$ we obtain $1800 \cdot 30/268 \approx 201.5$ tuples, which is the traditional cardinality estimation for the given query.

We now introduce two new estimation strategies, called $MIN$ and $MAX$, which make “extreme” assumptions about the underlying data distributions. Using $MIN$ and $MAX$, we do not reduce the frequency of all tuples in histogram $H(S.s)$ uniformly. Instead, we selectively choose which tuples in $S$ satisfy the condition $S.a < 10$ so that the resulting cardinality estimation is the smallest (for $MIN$) or largest (for $MAX$) possible.
CHAPTER 6. RECOMMENDING SITS

Using the MAX strategy, we choose the 30 tuples from $S$ that satisfy $S.a < 10$ in $H(S.s)$ as follows. First we pick the 8 tuples in $H(S.s)$’s third bucket, because each of these tuples joins with the largest number of tuples in $R$. Then, we pick 22 out of the 40 tuples in $H(S.s)$’s first bucket that match 10 tuples in $R$ each. If we now estimate the cardinality of the join using the original $H(R.s)$ and the propagated $H(S.s)$ we obtain $8 \times 50 + 22 \times 10 = 620$ tuples. Similarly, using the MIN strategy we proceed as follows. We first pick the 20 tuples in $H(S.s)$’s first bucket that do not join with any tuple in $R$. Then, we select 10 out of the 200 tuples in $H(S.s)$’s second bucket. Using the propagated $H(S.s)$ and the original $H(R.s)$ we estimate the cardinality for the query as $20 \times 0 + 10 \times 5 = 50$ tuples in this case.

|     | $\text{Card}(S.a<10, R \Join S)$ | $\text{Sel}(S.a<10, R \Join S)$ | $\text{Sel}(S.a<10|R \Join S)$ |
|-----|----------------------------------|---------------------------------|--------------------------------|
| MIN | 50                               | 0.00103                         | 0.0277                         |
| Traditional | 201.5                          | 0.00417                         | 0.1119 (=Sel(S.a<10))          |
| MAX | 620                              | 0.01285                         | 0.3444                         |

Figure 6.6: Cardinality and selectivity estimation using different strategies.

In general, the procedure to select $n$ tuples for strategy MIN (respectively, MAX) consists of sorting the set of values $\{2 \cdot 20 \cdot S-10 R, 1 \cdot 20 \cdot S-0 R, \ldots\}$ at the top of Figure 6.5 by increasing density values in $R$, and then selecting the first (respectively, last) $n$ tuples (not groups) in $S$ from the sorted list. This procedure effectively chooses the set of tuples in $S$ that minimize (respectively, maximize) the size of the resulting join.

Figure 6.6 summarizes the results discussed so far. Specifically, the first column in the figure shows the cardinality of the original query when using MIN, MAX, and the traditional strategy, given that the cardinality of $\sigma_{S,a<10}(S)$ is 30. The second column reports the selectivity of the query for each strategy, which is obtained by dividing the corresponding cardinality by the value $\mid R \times S \mid = \mid R \mid \cdot \mid S \mid = 180 \cdot 268$. Finally, the last column shows the conditional selectivity value $\text{Sel}(S.a < 10|R \Join S)$ for each alternative, defined as $\text{Sel}(S.a < 10, R \Join S)/\text{Sel}(R \Join S)$ and obtained by dividing the value in second column by $\text{Sel}(R \Join S) = \mid R \Join S \mid / \mid R \times S \mid = 1800/(180 \cdot 268)$. 
In the figure, the value of $\text{Sel}(S.a < 10 | R \bowtie S)$ for the traditional strategy (middle row) is 0.1119. Not surprisingly, this value is the same as $\text{Sel}(S.a < 10) = 30/268$. That is, $\text{Sel}(S.a < 10 | R \bowtie S) = \text{Sel}(S.a < 10)$ because the traditional strategy assumes independence. The two remaining values in the third column in the figure specify that $\text{Sel}(S.a < 10 | R \bowtie S)$ might take any value in the range $[0.0277, 0.3444]$ under the containment assumption, provided that $\text{Sel}(S.a < 10) = 0.1119$. In other words, those values indicate the potential difference in selectivity (introduced by predicate $R \bowtie S$) between $H(S.a)$ and $H(S.a | R \bowtie S)$ in the range $S.a < 10$.

We note that both the MIN and MAX strategies can be easily extended to handle filter predicates as well. For instance, consider query $\sigma_{S.a < 10 \land S.b > 5}(S)$ and let $S_a = |\sigma_{S.a < 10}(S)|$ and $S_b = |\sigma_{S.b > 5}(S)|$. Traditionally, after estimating $S_a$ using histogram $H(S.a)$, we uniformly reduce the bucket frequencies in $H(S.b)$ so that the total number of tuples in $H(S.b)$ is $S_a$. Instead, using MAX we choose from $H(S.b)$ as many tuples as possible that also satisfy $S.b > 5$, and therefore the resulting cardinality is $\text{min}(S_a, S_b)$. Similarly, for MIN we choose from $H(S.b)$ as many tuples as possible that do not simultaneously satisfy $S.b > 5$, and therefore the cardinality of the query is estimated by the expression $\text{max}(0, S_a + S_b - |S|)$.

**From MIN and MAX to $F_{\text{MIN}}$ and $F_{\text{MAX}}$**

It is important to note that in the example of Figure 6.5, the choice of tuples in $H(S,s)$ using strategies MIN and MAX does not depend specifically on predicate $S.a < 10$. Instead, it only depends on the number of tuples in $S$ satisfying $S.a < 10$. In fact, we obtain the same result if we replace $S.a < 10$ with any other predicate over $S$ that is satisfied by the same number of tuples. There is a one-to-one correspondence between the number of tuples that we select from $S$ and the number of tuples in the resulting join when using MIN and MAX, respectively.

Based on this observation, we define two functions, $F_{\text{MIN}}$ and $F_{\text{MAX}}$, that explicitly model this correspondence. $F_{\text{MIN}}$ and $F_{\text{MAX}}$ take as input a predicate $q$, a table $T$ men-
tioned in $q$, and the fraction $s$ of tuples that satisfy some (unspecified) predicate $p$ over $T$. In turn $F_{\text{MIN}}(q, T, s)$ (respectively, $F_{\text{MAX}}(q, T, s)$) returns the smallest (respectively, largest) conditional selectivity value $0 \leq \text{Sel}(p|q) \leq 1$ consistent with the given information $\text{Sel}(p) = s$ (using for that purpose the procedure described earlier). It is fairly simple to show that $F_{\text{MIN}}(q, T, 0) = F_{\text{MAX}}(q, T, 0) = 0$, $F_{\text{MIN}}(q, T, 1) = F_{\text{MAX}}(q, T, 1) = 1$, and $F_{\text{MIN}}(q, T, s) \leq F_{\text{MAX}}(q, T, s)$ for all predicates $q$, tables $T$, and selectivity values $s$. It follows that, for given $q$ and $T$, $F_{\text{MIN}}$ and $F_{\text{MAX}}$ define a closed region in $[0..1] \times [0..1]$, as illustrated in the following example.

**Example 25** Consider tables $R$ and $S$, as well as predicate $R \bowtie_{R,s=S,s} S$. Suppose that the join predicate corresponds to a foreign-key join with no referential integrity. That is, each tuple from $R$ matches with at most one tuple in $S$, and some (rare) tuples in $R$ have no match in $S$. Additionally, suppose that the join is skewed, with a few tuples in $S$ matching many tuples in $R$, and the remaining tuples in $S$ matching few tuples in $R$ or none at all. Figure 6.7(a) illustrates the functions $F_{\text{MIN}}(R \bowtie S, R, x)$ and $F_{\text{MAX}}(R \bowtie S, R, x)$ for $0 \leq x \leq 1$. Since $R \bowtie S$ is a foreign-key join, almost every tuple from $R$ matches with one tuple in $S$. Therefore, if we choose any $k$ tuples from $R$, the minimal and maximal number of tuples in the resulting join would be very close to $k$. This implies that $F_{\text{MIN}}(R \bowtie S, S, x) \approx F_{\text{MAX}}(R \bowtie S, S, x) \approx x$ for every $0 \leq x \leq 1$ and therefore the area between the two functions in Figure 6.7(a) is rather small. In contrast, Figure 6.7(b) illustrates functions $F_{\text{MIN}}(R \bowtie S, S, x)$ and $F_{\text{MAX}}(R \bowtie S, S, x)$ for $0 \leq x \leq 1$. Since $R \bowtie S$ is biased, a small number of tuples from $S$ might match many tuples from $R$. In this case, the minimal and maximal number of tuples in the join might be considerably different than when the independence assumption is used. Therefore, the area between $F_{\text{MIN}}(R \bowtie S, S, x)$ and $F_{\text{MAX}}(R \bowtie S, S, x)$ is much larger than that of Figure 6.7(a).

---

3If the foreign-key join verifies referential integrity, then $F_{\text{MIN}}(R \bowtie S, S, x) = F_{\text{MAX}}(R \bowtie S, S, x)$ for all $x$ and therefore the area between the functions is zero.
(a) \( F_{\text{MIN}}(R \bowtie S, R, x) \) and \( F_{\text{MAX}}(R \bowtie S, R, x) \) for \( 0 \leq x \leq 1 \).

(b) \( F_{\text{MIN}}(R \bowtie S, S, x) \) and \( F_{\text{MAX}}(R \bowtie S, S, x) \) for \( 0 \leq x \leq 1 \).

Figure 6.7: The area between \( F_{\text{MIN}} \) and \( F_{\text{MAX}} \) indicates the possible change in selectivity introduced by a given predicate.
CHAPTER 6. RECOMMENDING SITS

Formally, for a given predicate $q$ and table $T$ referenced in $q$, the area between $F_{\text{MIN}}$ and $F_{\text{MAX}}$ is defined as:

$$\text{Area}(q, T) = \sum_{0 \leq x \leq 1} \left( F_{\text{MAX}}(q, T, x) - F_{\text{MIN}}(q, T, x) \right)$$

The value $\text{Area}(q, T)$ is a number between 0 and 1, and indicates the average potential deviation from the independence assumption that is introduced by predicate $q$. A value of $\text{Area}(q, T)$ close to zero indicates that predicate $q$ is in general conditionally independent of other predicates applied to $T$, while values of $\text{Area}(q, T)$ close to one indicate that $q$ might introduce significant changes of selectivity.

Using $\text{Area}$ to Approximate $\text{diff}$ Values

The value $\text{Area}(q, T)$ indicates the average departure from independence introduced by predicate $q$ for a given selectivity in table $T$. Obtaining average values has two advantages for our problem. First, we can compute $\text{Area}(q, T)$ for each predicate once, and reuse that value during subsequent evaluations of error. Second, we can combine values of $\text{Area}(q, T)$ easily to estimate the average departure from independence introduced by several predicates. We now show how we estimate $\text{diff}_H$ when using $H = \text{SIT}(a|Q')$ to approximate $S = \text{Sel}(p|Q)$, where $Q' \subseteq Q$. In particular, our estimation of $\text{diff}_H$ also depends on $S$, so we denote it by $\text{diff}_{H,S}$.

1. Form a graph $G$, where each node in $G$ is either $p(a)$ or a predicate in $Q$, and there is an edge between nodes $p_i$ and $p_j$ in $G$ if the predicates reference some common
table. As an example, Figure 6.8(a) shows the graph created for $Sel(S.a < 10|R.s = S.s, S.t = T.t, T.b = 2)$.

2. Starting from $p(a)$, obtain a spanning tree for $G$. After adding edge $(p_i \rightarrow p_j)$ to the spanning tree, associate with node $p_j$ the value $Area(p_j, T)$, where $T$ is the common table shared by $p_i$ and $p_j$ (see Figure 6.8(b)). If there are multiple alternatives to extend the current spanning tree, choose the one with the largest $Area$ value.

3. Calculate $diff_{H,S}$ as follows:\footnote{There are alternative procedures to combine individual values of $Area(p, T)$. In our experiments, the alternative presented above resulted in good approximation of $diff$ values.}

$$
diff_{H,S} = \frac{\sum_{q_j \in Q'} Area(q_j, T)}{\sum_{q_j \in Q} Area(q_j, T)}
$$

This procedure returns $\frac{\text{Area}(R.s=S.s)}{\text{Area}(R.s=S.s) + \text{Area}(S.t=T.t) + \text{Area}(T.b=2)}$ for the example of Figure 6.8.

In other words, to estimate the $diff$ value for $\text{SIT}(a|Q')$ approximating $Sel(p(a)|Q)$ we analyze each predicate $q$ in $Q$ and we quantify how much each predicate alters incoming selectivity values. Then, we define $diff$ as the ratio between the sum of $Area$ values for the predicates in $Q'$ and the sum of all $Area$ values.

In summary, in this section we presented a technique to approximate $Diff$ values, which in turn can be used to estimate the expected error of the best decomposition of a given query, and in general the aggregated and weighted expected error of a given configuration $C$ with respect to the given workload $W$, that is, $error(C, W)$.

### 6.2.3 SIT Selection

In the previous section we showed how to obtain $error(C, W)$ for a given workload $W$ and arbitrary configurations $C$ that consist of candidate (not necessarily materialized) SITs. The smaller the value $error(C, W)$, the more desirable configuration $C$ becomes.
for workload \( W \). To recommend \( k \) SITs, we can in principle enumerate each possible configuration with \( k \) SITs and then return the one that minimizes \( \text{error}(\mathcal{C}, W) \). Unfortunately, the number of such configurations is large. For instance, in our experiments the number of candidate SITs was around 25,000 for a typical workload consisting of 200 queries. To obtain the top-20 SITs, we would need to evaluate \( \binom{25,000}{20} \approx 3.71 \cdot 10^{69} \) possible configurations, which is of course infeasible in practice.

Instead, we adopted a simple greedy strategy to recommend SITs. The procedure to recommend \( k \) SITs consists of \( k \) iterations. During the \( i \)-th iteration, we calculate the value \( \text{error}(\mathcal{C}_W \cup \{H_1, \ldots, H_{i-1}\} \cup \{H\}, W) \) for each candidate SIT \( H \), where \( \{H_1, \ldots, H_{i-1}\} \) are the best SITs obtained in iterations 1 to \( i - 1 \), and \( \mathcal{C}_W \) is the initial configuration used when executing the queries in \( W \). Next, we identify the candidate histogram with the minimal expected error and add it to the current configuration. Note that we recompute \( \text{error} \) values after each iteration, because the presence of a new SIT in \( \mathcal{C} \) usually causes some decompositions (and thus some \( \text{error} \) values) to change. To compute \( \text{error} \) values, we need to obtain the best decomposition for each \( sp^i_q \) using SITs \( \mathcal{C}_W \cup \{H_1, \ldots, H_{i-1}\} \cup \{H\} \). For that purpose, we use algorithm \( \text{getSelectivity} \) of Chapter 4. In this way, by using the same algorithm during both the recommendation and the actual cardinality estimation phases, we can guarantee that the recommended SITs will be used during cardinality estimation. Figure 6.9 summarizes this procedure to choose the top-\( k \) SITs for a given workload \( W \).

In general, our technique assumes that SITs do not strongly interact with each other. For instance, in the context of automatic index selection, reference \([CN97]\) uses a “greedy(\( m, k \))” procedure, in which an optimal configuration of \( m \) indexes is initially identified by using exhaustive search, and then such configuration is greedily extended until it contains \( k \) indexes. The reason is that sometimes two or more indexes that are not useful individually can have large impact during execution if used together. In such cases, a pure greedy technique will miss those alternatives. In contrast, in our scenario there are no such interactions between SITs. That is, the overall increase in accuracy given by two
recommend_SITs (W: workload, k: number of SITs to materialize)
returns (C: set of k SITs to materialize)
01 \( S = \) set of all candidate SITs for \( W \)  // Section 6.2.1
02 \( C = \emptyset \)
03 while \( |C| < k \)
04 \( H_{\text{best}} = \) NULL; \( \text{err}_{\text{best}} = \infty \)
05 for each \( H \in S \)
06 \( \text{err} = \text{error}(\hat{C}_W \cup C \cup \{H\}, W) \)  // Section 6.2.2
07 if (\( \text{err} < \text{err}_{\text{best}} \))
08 \( H_{\text{best}} = H; \text{err}_{\text{best}} = \text{err} \)
09 \( C = C \cup \{ H_{\text{best}} \} \)
10 \( S = S - \{ H_{\text{best}} \} \)
11 return \( C \)

Figure 6.9: Recommending SITs for a workload.

SITs is usually a linear combination of the individual increase in accuracy due to each SIT in isolation (due to the aggregate function \( E \) in algorithm getSelectivity). Although more complex and thorough search strategies are possible, the proposed technique performed sufficiently well in our experiments (see Section 6.3).

In our implementation we use some optimizations over the basic procedure of Figure 6.9. For instance, after adding a new SIT to \( C \) in line 7, we mark all the queries in \( W \) that actually exploit such SIT, and in the next iteration we just recompute the scores for those queries in lines 4-5 (error values for the rest of the queries remain unchanged). We also extended getSelectivity to incrementally obtain the best decomposition when a single new SIT, \( H_i \), is added in line 5.

### 6.3 Experimental Evaluation

In this section we present an experimental evaluation of an implementation of the algorithm for recommending SITs over a commercial RDBMS. Specifically, we modified the server code of Microsoft SQL Server 2000 to include the framework of Section 4.2 to exploit SITs, and implemented the algorithm of Section 6.2 as a client connecting to the server via ODBC. We used the internal histograms of Microsoft SQL Server to implement
SITs. Each SIT was given at most 200 buckets worth of space and was created using the same algorithm that Microsoft SQL Server uses to build base-table histograms, which is a variation of MaxDiff.

6.3.1 Results

For our experiments we used the same database and workload generator from Section 4.3. Specifically, for each experiment we generated 400 random SPJ queries consisting of three to seven join predicates and three additional filter predicates. We then divided the 400 queries into two workloads of 200 queries each, which we call the training and validation workloads, respectively.

6.3.1.1 Evaluating SIT Recommendations

In this experiment we first optimized and evaluated all queries in both the training and validation workloads using base-table statistics but no SITs for cardinality estimation. We then obtained the top-\(k\) recommended SITs for different values of \(k\), using the training workload as the input of our algorithm of Section 6.2. We materialized each recommendation, and then we optimized and evaluated the queries in both the training workload (to verify the usefulness of the recommended SITs), and in the validation workload (to test whether the recommended SITs can be exploited by queries that were not present during the recommendation phase).

Finally, we also evaluated the scenario in which all possible SITs were present during optimization. Instead of generating all possible SITs (over 28,000 for each workload) we proceeded as follows. We first optimized each query independently and intercepted every cardinality request by the optimizer. This way, we saved in a temporary file all sub-plans that were explored during the optimization of the queries in the workload. We then evaluated each sub-plan and obtained the correct cardinality estimation. Next, we optimized each query in the workload once more, this time replacing each cardinality request with the correct value that was previously calculated. In this second iteration, the
optimizer analyzed new sub-plans that were not previously explored and returned different execution plans for some queries: more accurate cardinality estimates changed the search strategy of the optimizer, and therefore new portions of the search space were reached. We again saved to disk the new sub-plans and obtained the correct cardinality estimates. We sequentially performed the steps of optimization and cardinality estimation until no new sub-plans were explored by the optimizer. At that point, the optimizer obtained the correct cardinality estimation for every explored sub-plan, which is equivalent to the case in which every possible SIT is available. In our experiments, three iterations were sufficient before optimization stabilized.

Figure 6.10: Evaluation of the top-$k$ recommended SITs for varying $k$. 
Figures 6.10(a) and 6.10(b) show the execution times for the training and validation workloads, respectively, when different sets of recommended SITs are available during cardinality estimation. When no SITs are available, the training workload was executed in 1691 seconds, and the validation workload was executed in 1452 seconds (those values are shown in the figures as recommendations of size zero). On the other extreme, when all SITs are available to the optimizer, the resulting execution times were 499 and 488 for the training and validation workloads, respectively, which represents a three-fold improvement in execution time.

Figure 6.10 shows that our technique to recommend SITs rapidly converges to configurations that are virtually indistinguishable from the optimal ones. For instance, by just materializing the top-5 SITs we obtained execution times that are on average around 30% more efficient than when no SITs are available. In fact, for 20 or more recommended SITs, the resulting execution times for both the training and validation workloads are the same than when all possible SITs are available. In general, we observed that even when using 40 SITs the cardinality estimates for the sub-plans are not perfect. However, the inaccuracies are sufficiently small that the optimizer makes the same choices as if the correct estimates were used.

Figure 6.11(a) contrasts the execution time of each query in the validation workload when the top-30 recommended SITs are materialized (x-axis) against the corresponding execution times when no SIT is available (y-axis). We see that almost all queries resulted in more efficient executions, and in some situations the improvement in execution time is over an order of magnitude. Figure 6.11(b) shows a close-up of the same data for queries that executed in 0 to 8 seconds. The figure still shows that the majority of queries had lower execution times. However, it also shows a small group (around 10 queries) whose execution times were up to 13% slower than when no SITs where available. We closely examined these cases and found that sometimes the execution plans were the same for both scenarios. However, some internal parameters of the plan operators were different (e.g., the amount of memory assigned to each operator). In other situations,
Figure 6.11: Execution time of validation queries optimized with no SITs vs. with the top-30 SITs.
the plans were slightly different (e.g., some join algorithms were replaced by others). In general, for all such instances the optimizer obtained significantly better cardinality estimates when the top-30 SITs where available. However, as explained in Section 6.1, the estimated cost of an execution plan uses additional information that might introduce (minor) inaccuracies to the cost estimates. The 10 queries over the $x = y$ diagonal in Figure 6.11(b) are examples of such situations.

The proposed technique tends to identify simple SITs that are common to many queries and some specific SITs that can be exploited in expensive queries. For that reason, the resulting recommendations can be used to obtain better estimates for queries not only in the training workload, but also in the validation workload.

6.3.1.2 Evaluating the What-if Component

In this section we evaluate the different components of the $error$ function used in the what-if module to rank alternative SIT configurations. Recall from Section 6.2.2 that the $error$ of configuration $C$ with respect to workload $W$ is defined as:

$$error(C, W) = \sum_{q \in W} execTime_q \cdot error(C, q)$$

where

$$error(C, q) = \sum_{sp_i_q} \left| est_i_q - act_i_q \right| \cdot \frac{error(C, sp_i_q)}{error(C_W, sp_i_q)}$$

and $error(c, p)$ is the expected error of the most accurate decomposition –returned by $getSelectivity$– to estimate the cardinality of $p$ using configuration $c$. We evaluate the individual components of the definition of $error(C, W)$ by comparing the following alternatives:

- **full**: This is the proposed technique of Section 6.2.3 that uses both feedback from previous query executions and the techniques of Section 6.2.2.1 to approximate $error(C, sp_i_q)$ values.

- **no-feedback**: This strategy does not exploit feedback from previous executions of the workload queries. For this reason, we redefine the error of a configuration with
respect to a given workload as:

$$\text{error}(C, W) = \sum_{q \in W} \text{estimated\_execution\_time}(C_W, q) \cdot \text{error}(C, q)$$

In other words, we replace the actual execution time of each query by the estimated execution time for $q$ obtained when the RDBMS originally optimized and evaluated the queries in the workload using configuration $C_W$. Also, we redefine the error of a configuration with respect to a query in the workload as:

$$\text{error}(C, q) = \sum_{\text{sp}_q^i} \frac{\text{error}(C, \text{sp}_q^i)}{\text{error}(C_W, \text{sp}_q^i)}$$

In other words, we replace the factor $|\text{est}_q^i - \text{act}_q^i|$, which represents the observed error in estimation for each sub-plan est$_q^i$, by the constant one.

- **minimal**: Just as in the no-feedback technique, the minimal strategy does not exploit feedback from the execution of previous queries. Additionally, minimal uses a simpler definition of $\text{error}(C, \text{sp}_q^i)$. Specifically, we replace $\text{error}(C, \text{sp}_q^i)$ with $nInd$, the alternative “syntactic” definition of Chapter 4. $nInd$ simply counts the number of independence assumptions made when approximating a selectivity value with a given SIT, and therefore does not require such SIT to be materialized.

Figure 6.12 shows actual execution times—for the training and validation workloads—when using the top-$k$ recommended SITs by error variations in the what-if module, and for different values of $k$. In general, minimal, which is the simplest strategy, converges to the best possible configuration slower than the other techniques. (In our experiments we needed over 80 SITs to obtain results similar to those for full and 20 SITs.) One reason is that minimal uses $nInd$ to model the impact of candidate SITs, which results in similar problems to those reported in Chapter 4: many candidate SITs share the same value of $nInd$, and ties are broken arbitrarily. In contrast, no-feedback approximates Diff using the technique of Section 6.2.2.1 and therefore produces better recommendations than minimal, especially for 20 or more SITs.
Neither minimal nor no-feedback exploit feedback from previous executions to weight the impact of candidate SITs. Instead, all query sub-plans are assigned the same importance within a query, and each query is weighted by the estimated execution cost. This procedure introduces some inaccuracies because the estimated cost of the original queries is usually not a good indicator of actual execution times (because relevant SITs are missing when estimating those costs). However, both minimal and no-feedback still perform reasonably well and can be used in the absence of query feedback.
Finally, Figure 6.12 shows that, as expected, full always results in the best recommendations because it guides the choice of SITs using more information than the other variants. Specifically, for 5 and 10 queries, full’s execution plans are around 30% more efficient than those produced by no-feedback and minimal. For larger recommendations, the trend is maintained but the gap between the strategies is reduced.

6.4 Conclusion

In this chapter we introduced a technique to recommend small but useful sets of unidimensional SITs with SPJ generating queries in an RDBMS. Our algorithms guide the selection of SITs by exploiting workload information and feedback from the execution of previous queries, and obtain recommendations without materializing any SIT in advance. The main contributions of this chapter are summarized below:

- We designed a what-if module to evaluate the impact of SITs that are not yet materialized. For that purpose, we proposed a technique based on two novel extreme strategies to estimate cardinality values, MIN and MAX, which quantifies the average difference in distribution between a candidate SIT($a|Q$) and the histogram built over the corresponding base table $H(a)$.

- We designed a technique to rank the set of candidate SITs to materialize without building them in advance. The proposed technique works closely with the optimizer, hence assuring that the recommended SITs will be used by the optimizer during subsequent cardinality estimation requests.

- We reported experimental results over an implementation of the algorithms in this chapter (and also the framework of Chapter 4) over Microsoft SQL Server.
Chapter 7

Related Work

In this chapter we discuss the literature that is relevant to this thesis. We start in Section 7.1 by reviewing some notable efforts in the literature of relational query optimization (Chapter 2). Then, in Section 7.2 we discuss recent work that is related to the concept of SITs (Chapters 3 and 4). We next review in Section 7.3 bibliography that is relevant to the SIT’s creation problem (Chapter 5). Finally, in Section 7.4 we discuss techniques that share the motivation of Chapter 6 and provide different types of recommendations to improve the overall performance of relational database systems.

7.1 Query Optimization

Current relational optimizers are influenced by techniques introduced in the System-R query optimizer [SAC+79]. One important contribution of this reference is a cost-based framework to obtain execution plans, which is still used with some variations in most current optimizers (see Chapter 2 for more information on this topic and on different statistical estimators used to approximate costs). Another important contribution of [SAC+79] is a bottom-up dynamic programming search strategy to traverse the space of candidate execution plans. This strategy needs to consider $O(3^n)$ expressions [OL90] for a given SPJ query. To decrease optimization time, some heuristics are used such as delaying the optimization of cartesian products, or considering only left-deep join trees.
CHAPTER 7. RELATED WORK

The Starburst optimizer [HFLP89, HCL+90] extends System-R with a more efficient and extensible approach, and consists of two rule-based subsystems. Initially, a Query Graph Model (QGM) optimizer uses a set of production rules to heuristically transform the abstract representation of the input query into a “better” alternative (this stage does not use cost information and therefore optimality can be compromised). In the second phase the actual execution plan is chosen. For this, physical operators (called LOLEPOPs) can be combined in many ways to implement higher-level operators, and such combinations are expressed in a grammar production-like language [Loh88]. The join enumerator in Starburst is similar to System-R’s bottom-up enumeration scheme.

The Exodus optimizer generator [GD87] is the first extensible optimization framework that uses a top-down approach. Exodus separates the optimizer’s search strategy from its data model, and distinguishes between transformation rules (which map one algebraic expression into another) and implementation rules (which map an algebraic expression into an operator tree). Although it was difficult to construct efficient optimizers using Exodus, it provided a useful foundation for the next generation of extensible optimizers.

The Volcano Optimizer Generator [GM93] improves the efficiency of Exodus and introduces more extensibility and effectiveness. Volcano’s search algorithm combines dynamic programming with directed search based on physical properties, branch-and-bound pruning and heuristic guidance. Finally, the Cascades framework [Gra95] solves some problems present in Exodus and Volcano, and improves functionality, ease of use, and robustness without compromising extensibility and efficiency. Cascades is the state-of-the-art rule-based optimization framework used in current optimizers such as Tandem’s NonStop SQL [Cel96] and Microsoft SQL Server [Gra96]. The Cascades framework differs from Starburst in its approach to enumeration. In fact, this system does not use two distinct optimization phases as Starburst does, and the application of rules is goal-driven, as opposed to the forward-chaining rule application phase in Starburst. A detailed description of Cascades and some extensions to the original framework appear in [Xu98, Bil97].
7.2 Statistics on Query Expressions

The idea of building statistics over non-base tables first appears in [AGPR99]. This reference introduces join synopses, which are pre-computed samples of a small set of distinguished joins. Joins must be defined between foreign and primary keys, and therefore a single sample for each table is enough to provide approximate answers for a large number of queries. The idea is to conceptually materialize the extended table obtained by applying all foreign-key joins, and then take a uniform sample over this result. Reference [GLR00] extends this approach by introducing the concept of icicles, which are a new class of samples that tune themselves to a dynamic workload. Intuitively, the probability of a tuple being present in an icicle is proportional to its importance for answering queries in the workload. In general, once samples are obtained, input queries can be rewritten to use the samples instead of the corresponding base tables. Therefore, by operating on the sample domain, we obtain (approximate) answers using just a fraction of the original execution time. In contrast, SITs as introduced in this thesis can be defined over arbitrary query expressions and require significantly fewer resources than samples. The reason is that we are interested in cardinality estimation during optimization rather than providing approximate answers of user queries.

Similar to previous work in self-tuning histograms [AC99, BCG01a], LEO (DB2’s LEarning Optimizer) [SLMK01] is an online framework that repairs incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed queries, LEO computes adjustments to base-table statistics, which can then be used during the optimization of subsequent queries. This procedure is similar to what we propose in Chapter 3 for SITs. The key difference with our approach is that LEO maintains a single adjusted histogram per attribute and still relies on the independence assumption during cardinality estimation. However, the adjustments are done in such a way that the cardinality of the processed query is correctly calculated despite assuming independence. For instance, consider query $\sigma_{S.a<10}(R \bowtie S)$, and suppose that the chosen execution plan evaluates the selection after the join. Further, assume that originally the selectivity of
CHAPTER 7. RELATED WORK

The predicate $S.a < 10$ is estimated as 0.1, and the selectivity of the join predicate as 0.25. Then, the cardinality of the query is estimated as $|R| \cdot |S| \cdot 0.025$. Now, suppose that when analyzing feedback from the actual execution of the query LEO finds that the selectivity of the join is indeed 0.25, but the fraction of tuples from $R \bowtie S$ that satisfy $S.a < 10$ is much larger than originally estimated, say, 0.5. In this situation, the actual cardinality is $|R| \cdot |S| \cdot 0.125$, which is considerably different from the original estimation. To address this mismatch, LEO adjusts the histogram $H(S.a)$ so that the same range query returns the value 0.5, and therefore, the cardinality of the query is computed accurately. A drawback of this approach is that a single adjusted histogram is maintained for each attribute in the system. However, the same attribute can have radically different distributions depending on — using SIT’s terminology — its generating query. (See Figure 3.2 in Chapter 3 for an example of two different distributions for the same attribute, depending on the specific context.) For this reason, LEO’s approach may result in oscillations and does not converge to a stable statistical summary. As a consequence, during the optimization of the query in the example above, the same adjusted histogram $H(S.a)$ is used to estimate both $Sel_S(S.a)$ and $Sel_{R,S}(S.a|R \bowtie S)$ using the notation of Chapter 4. If both distributions are different, the adjusted histogram incurs large estimation errors for at least one alternative. SITs avoid this fundamental problem by relying on different statistics for the same attribute depending on its particular context in the corresponding query plan.

Reference [GTK01] shows how to use probabilistic relational models to approximate the joint distribution of multiple attributes in one table, or in different tables that are combined using foreign-key joins. Probabilistic relational models (PRMs) are a language to compactly represent complex joint distributions over high-dimensional spaces, and they extend statistical models such as Bayesian networks to the relational domain. Reference [GTK01] uses the concept of conditional independence between attributes to decompose the representation of a join distribution into factors that capture the actual independencies that hold in the data domain, therefore obtaining a compact representation of the actual distribution. In [GTK01], a PRM is first constructed for the database.
Then, the resulting PRM is used to estimate the cardinality of input SPJ queries with foreign-key joins. This work mostly focuses on point queries such as \( a_1 = v_1 \land \ldots \land a_n = v_n \) and assumes that each attribute has a small discrete domain. It also presents some extensions to relax these assumptions, but it is not clear how the resulting techniques can be implemented efficiently. With a similar motivation, reference [DGR01] uses statistical interaction models to explicitly identify and exploit the statistical characteristics of the underlying data. The rationale is that real tables are characterized by complex correlation patterns, where a certain subset of the attributes can be (unconditionally) independent of another attribute subset, or, alternatively, can be (conditionally) independent of given a third subset of attributes. The idea in [DGR01] is to break the statistics (e.g., multidimensional histograms) into (i) an interaction model that accurately captures significant correlation and independence patterns in data, and (ii) a collection of lower-dimensional histograms that, based on the model, can provide accurate approximations of the overall joint data distribution. We believe that the ideas introduced in [GTK01] and [DGR01] can be incorporated into our framework of Section 4. If we can infer from the data distribution that some predicates are conditionally independent of others, we can apply semantic versions of the “separable decomposition” property to further reduce the space of decompositions. As a simple example, consider \( \text{Sel}_R(p_1, p_2, p_3) \). If we can infer that \( p_1 \) is conditionally independent of \( p_2 \) given \( p_3 \) using the techniques described above, we can decompose the given selectivity value as \( \text{Sel}_R(p_1, p_2, p_3) = \text{Sel}_R(p_1 | p_2) \cdot \text{Sel}_R(p_2, p_3) \) without relying on any assumption, such as independence.

### 7.3 Building SITs

Sampling has been proposed in the literature as a primitive relational operation [OR90]. Reference [CMN99] studies how to commute sampling and join operators. The main problem is that the join of random samples of tables \( R \) and \( S \) is not a sample of the join \( R \bowtie S \). For that reason, [CMN99] introduces several algorithms to sample the result of a join operation without computing the entire join in the first place. The key difference
with our approach is that \cite{CMN99} produces an actual sample of the result. For this reason, these techniques actually evaluate joins (at least in restricted ways), which can be expensive. In our scenario, when we create $\text{SIT}(S.a|R \bowtie S)$ we are not interested in actual values from table $R$, since we only need an approximate distribution of $\pi_{S.a}(R \bowtie S)$. Therefore, we can obtain multiplicity values for all tuples from $S$ and sample over the intermediate result that approximates (modulo containment) the full table $\pi_{S.a}(R \bowtie S)$, which can improve accuracy without increasing execution times. Other references that also address the problem of sampling over joins are \cite{HNS94,GGMS96}, but they mainly focus on selectivity estimation of queries with join predicates. Finally, we mention some work that focuses on estimating aggregate queries using sampling techniques \cite{AGP00}, possibly exploiting workload information \cite{CDD01,CDN01}. We can see a histogram as a special “fuzzy” group-by summary, where the groups are not syntactically determined, but instead based on a similarity in their distributions. The main difference between our problem and that of the above references is that they use a different error metric to materialize the samples. In those references, each group must have an accurate estimation, and therefore the focus is to bias the sample towards groups with very few items. Those references also consider just the single-table scenario for aggregated queries, or some simple extensions with foreign-key joins.

Multi-query optimization aims at exploiting common sub-expressions in the input queries to reduce evaluation cost. It has been observed that exhaustive algorithms for multi-query optimization are impractical, since they explore a doubly exponential search space on the query sizes. Reference \cite{RSSB00} proposes cost-based heuristics that can be incorporated to existing optimizers, and shows that the algorithms provide significant benefits over traditional optimization with acceptable overhead. In Chapter 5 we address a more constrained optimization problem, since the execution plans for each individual SIT is one of few alternatives known in advance. For this reason, the search space is much smaller than in the general case, and we are able to adapt SCS to find the optimal global strategy. The Shortest Common Supersequence (SCS) is a well studied problem
in the literature [Mai78]. Finding the SCS of a set of sequences is NP-complete, and can be solved by dynamic programming in $O(l^n)$ time for $n$ input sequences of length $l$ [Tim90]. Reference [NO01] introduces an A* algorithm to solve SCS, which is generally more efficient than the classic dynamic programming technique. In Chapter 5 we modified such algorithm so that it fits our generalized problem of finding an optimal schedule to create a set of SITs.

### 7.4 Recommending SITs

Our techniques in Chapter 6 that recommend SITs in an RDBMS are inspired by previous work on automatic physical database design [CN97, ACN00, ACDN03]. These references present workload-based approaches to recommend indexes, materialized views, and disk layouts for tables and auxiliary data structures in an RDBMS, so that subsequent queries drawn from similar workloads result in improved execution times. While our specific problem is fairly different from those presented in the references above, we adapted some of the main ideas introduced earlier to our scenario. For instance, we use information about given workloads to guide the selection of SITs. We also rely on a what-if component [CN98] to evaluate the impact of hypothetical configurations in the RDBMS. Finally, analogously to [ACN00], our techniques work closely with the actual cardinality estimation module during SIT selection to guarantee that the recommended SITs are used during subsequent query optimizations.

Reference [CN00] introduces MNSA, a workload-based technique that identifies the minimal set of base-table statistics that need to be created in a database system to avoid sacrificing the quality of generated query plans. The procedures exploit a relaxed notion of plan equivalence, in which two plans $p_1$ and $p_2$ are $t$-Optimizer-Cost equivalent if the query optimizer predicts that the execution costs of $p_1$ and $p_2$ are within $t$ percent of each other. For a given workload, the MNSA algorithm incrementally identifies and builds new statistics over the base tables until it determines that no additional statistics are needed. To test whether the current subset of statistic is enough for estimation purposes, MNSA
considers how the presence of the remaining statistics would impact query optimization. The idea is to replace the magic numbers (used by the optimizer when no histogram is available) with extremely small and large values, and then test whether the resulting plans significantly change when the optimizer uses extreme estimations for attributes that have no associated base-table histograms. In Chapter 6 we adapt some of the ideas in MNSA by introducing functions $F_{\text{MIN}}$, $F_{\text{MAX}}$, and $\text{Area}$. These functions model the difference in cardinality estimation that we obtain on intermediate results when we make extreme assumptions about the underlying data distributions. To improve the quality of the resulting recommendations, in Chapter 6 we additionally rely on feedback from previously executed queries to guide our search. The top-level algorithm in [CN00] is different from our main technique. In fact, MNSA continues building base-table statistics until no new histogram would make the optimizer choose a significantly different execution plan. In our case, we provide mechanisms to rank the set of candidate SITs and materialize just the top-$k$ ones for a given value of $k$.

Reference [KW03] studies the problem of computing an optimal combination of histograms (that are derived from [KW99]) for a given workload consisting of SPJ queries and a limited amount of available memory. In particular, that work assumes that the cardinality of each query $q$ in the workload is estimated using exactly one histogram that captures all relevant attributes for $q$. In other words, for a SPJ query joining tables \{R$_1$, \ldots, R$_n$\} and having range predicates over attributes \{a$_1$, \ldots, a$_k$\}, the candidate set of (multidimensional) histograms to materialize for $q$ are those built on the result of $R_1 \bowtie \cdots \bowtie R_n$ over a superset of attributes \{a$_1$, \ldots, a$_k$\}. Using the notation of this thesis, these candidate histograms are all SIT($A|R_1 \bowtie \cdots \bowtie R_n$), where \{a$_1$, \ldots, a$_k$\} $\subseteq$ $A$. For efficiency purposes, two pruning techniques reduce the search space. The procedure in [KW03] to select a set of statistics first materializes all candidate histograms of all possible sizes over the actual database, and then it calculates the actual error of each alternative for the given workload. Once all error information has been collected, a dynamic programming technique finds the combination and sizes of candidate histograms
that minimizes the overall error for the input workload. One of the main differences with the approach of Chapter 6 is that we exploit multiple SITs to estimate the selectivity of any input query. In fact, we might still use a histogram that does not cover all relevant attributes of a query, by relying on some independence assumption. For that reason, the search space of candidate SITs is much larger than that of [KW03] (see Section 4.2.1.1 for bounds on the number of SITs that might be useful for a query). Therefore, in our scenario it is infeasible (and unrealistic) to build in advance all possible candidate statistics and extract some information from them to guide the recommendation process. In fact, we take the opposite approach: we try to infer which statistics are the most useful for cardinality estimation without building them beforehand.
Chapter 8

Conclusions

In this thesis we introduced SITs, a new source of statistical information for optimizers in relational database systems. SITs are statistics on query expressions and can be used to accurately model the distribution of tuples on intermediate nodes in query execution plans. We showed that when appropriate SITs are available during optimization, the resulting query plans are generally improved, with execution times that are just a fraction of those obtained when only base-table statistics are available.

We comprehensively studied the main obstacles to effectively incorporate SITs into existing database systems. We showed how to automatically identify useful SITs to build in a database system. We then studied the problem of efficiently creating such recommendations. Finally, we indicated how to exploit available SITs during cardinality estimation to produce better execution plans for input queries.

To conclude this thesis, we show in Figure 8.1 an integrated diagram of the different techniques, algorithms, and components introduced in previous chapters. Initially, each query posed to the RDBMS is parsed and transformed into an algebraic tree. This abstract representation of the input query is then passed to the optimizer, which finds an efficient execution plan to evaluate the query. For that purpose, the optimizer exploits the set of available SITs (if any) during cardinality estimation (Chapter 4). Then, the resulting query plan is passed to the execution engine, where it is evaluated. While result
tuples are returned, the RDBMS might use query feedback to refine some existing SITs, so that they become more accurate for subsequent queries (Section 5.1). In parallel, information about the query execution is saved for future use. After several queries are executed in this way, an off-line component uses the workload information previously saved to obtain a small set of additional SITs to materialize (Chapter 6). Those SITs are then created (see Section 5.2) and incorporated to the pool of available SITs in the RDBMS. Using this additional information, the RDBMS continues optimizing and evaluating new queries, and the cycle repeats.

We believe that SITs constitute a well-founded approach for dealing with complex data correlations during cardinality estimation, and can positively impact the efficiency of future relational database systems.
Bibliography


[Cel96] Pedro Celis. The query optimizer in Tandem’s new ServerWare SQL Product. In *Proceedings of the Twenty-second International Conference on Very Large Databases (VLDB’96)*, 1996.


[DGR01] Amol Deshpande, Minos Garofalakis, and Rajeev Rastogi. Independence is good: Dependency-based histogram synopses for high-dimensional data. In


BIBLIOGRAPHY


[PSTW93] Bernd-Uwe Pagel, Hans-Werner Six, Heinrich Toben, and Peter Widmayer. Towards an analysis of range query performance in spatial data structures. In


