Exploiting Constraint-Like Data Characterizations in Query Optimization

Parke Godfrey
York University
4700 Keele Street
Toronto, Ontario M3J 1P3
Canada
godfrey@cs.yorku.ca

Jarek Gryz
York University
4700 Keele Street
Toronto, Ontario M3J 1P3
Canada
jarek@cs.yorku.ca

Calisto Zuzarte
IBM Canada Ltd.
1150 Eglinton Ave. E.
Toronto, Ontario M3C 1H7
Canada
calisto@ca.ibm.com

ABSTRACT

Query optimizers nowadays draw upon many sources of information about the database to optimize queries. They employ runtime statistics in cost-based estimation of query plans. They employ integrity constraints in the query rewrite process. Primary and foreign key constraints have long played a role in the optimizer, both for rewrite opportunities and for providing more accurate cost predictions. More recently, other types of integrity constraints are being exploited by optimizers in commercial systems, for which certain semantic query optimization techniques have now been implemented.

These new optimization strategies that exploit constraints hold the promise for good improvement. Their weakness, however, is that often the "constraints" that would be useful for optimization for a given database and workload are not explicitly available for the optimizer. Data mining tools can find such "constraints" that are true of the database, but then there is the question of how this information can be kept by the database system, and how to make this information available to, and effectively usable by, the optimizer.

We present our work on *soft constraints* in DB2. A soft constraint is a syntactic statement equivalent to an integrity constraint declaration. A soft constraint is not really a constraint, per se, since future updates may undermine it. While a soft constraint is valid, however, it can be used by the optimizer in the same way integrity constraints are. We present two forms of soft constraint: *absolute* and *statistical*. An absolute soft constraint is consistent with respect to the *current* state of the database, just in the same way an integrity constraint must be. They can be used in rewrite, as well as in cost estimation. A statistical soft constraint differs in that it may have some degree of violation with respect to the state of the database. Thus, statistical soft constraints cannot be used in rewrite, but they can still be used in cost estimation.

ACM SIGMOD 2001 May 21-24, Santa Barbara, California USA

We are working long-term on absolute soft constraints. We discuss the issues involved in implementing a facility for absolute soft constraints in a database system (and in DB2), and the strategies that we are researching. The current DB2 optimizer is more amenable to adding facilities for statistical soft constraints. In the short-term, we have been implementing pathways in the optimizer for statistical soft constraints. We discuss this implementation.

1. INTRODUCTION

Integrity constraints (ICs) have long been the key means for expressing semantics in databases. ICs are declarative statements just as are queries written in SQL. Unlike queries however, an IC expresses a property that must be true about the database. Once declared, the database system is obliged to protect the integrity of the database as expressed by its ICs. The availability of a mechanism to express explicitly declarative, semantic characterizations in databases has proven useful for other purposes beyond ensuring database integrity as well [9]. In the 1980's, semantic query optimization (SQO) techniques were introduced which offer additional optimization opportunities by exploiting the database's ICs [17]. More recently, certain SQO techniques have been implemented and employed successfully in the optimizers of commercial systems [6, 24]. These techniques have been seen to offer tremendous cost improvements for certain types of queries in standard, common workloads and databases. We believe there is great promise in further development and deployment of SQO techniques.

A disadvantage of these techniques though is that the semantic characterizations they require are not always available as ICs associated with the database. This is not to say that these characterizations do not generally hold in real-world databases. Rather, there exists evidence that they often do. They rarely are, however, available explicitly as ICs. This is the case for several reasons.

1. Only recently have people started to use ICs in greater concentration as they seek to capture more business logic in the database, and hence, to place these business rules under the protection of the database system. The expense of integrity checking has always limited people's use of ICs.
2. Many potentially useful semantic characterizations true of a database simply are not known. If known, they

could potentially be expressed as ICs [13].

3. Even if a useful characterization is true, there may be no justification to make it an IC. That is, there is no reason to insist that this *remain* true, that it is a part of the database's *integrity*.

On point 1, database developers desire to express more business logic in the database. This must drive database research and development to provide greater expressiveness for ICs *and* greater optimization of constraint checking. On point 2, there has been successful data mining work as of late to discover useful constraint-like characterizations over databases. This work is focused on the discovery of useful "constraints" for use in query optimization [10]. On point 3, little work exists. There simply are not alternate mechanisms to express declarative constraint-like characterizations of a database aside from ICs.

Towards addressing point 3, we describe useful new mechanisms that we are implementing in DB2 for expressing semantic information in databases. We introduce new types of "constraint" information.

An *informational constraint* is an integrity constraint (IC) for which an external promise has been made that it will never be violated. Thus informational constraints never need to be (expensively) checked by the system, but they may be used by the optimizer just as any other IC. This addresses, in part, point 1 from above.

We changed the parser for DB2 to accommodate the declaration of informational constraints. Any of the ICs that DB2 supports can be marked as informational. The system has been adapted so that informational constraints are ignored for checking, but the optimizer uses informational constraints just as any other ICs.

Constructs similar to informational constraints exist in ORACLE-8 where the standard integrity constraints are marked with a *rely* option [23], so that they are not verified on updates. There are a number of environments in which informational constraints are beneficial. In many data warehousing environments, all data loading (so essentially all updating) is done by loader applications. These applications may check certain data integrity themselves. It is not necessary for the DBMS to recheck these same integrity constraints. Such constraints can be marked as informational to avoid checking. They are still available for the optimizer.

A *soft constraint* (SC), on the other hand, is an IC-like statement that is true with respect to the current state of the database, but which does not carry the same weight as an IC. The next update to the database could invalidate a soft constraint. If a transaction would invalidate an actual IC (a "hard" constraint), the transaction is aborted and the consistency of the database is preserved. If a transaction would invalidate a soft constraint, however, it is not aborted (for that reason). Instead, if the transaction commits, the SC is "aborted" since it is no longer consistent with the database. Thus consistency of the database is still maintained, but by different means. However, until a soft constraint becomes invalid, the optimizer can use it as if it were an IC. This addresses point 3 from above.

Clearly then, SCs are not meant to protect the integrity of the database as do ICs; but like ICs, they do semantically characterize the database. As certain types of ICs are now used in query optimization, SCs can be used in the optimizer in the same way. If there are any useful characterizations of the database valid with respect to the current state of the database and useful for the optimizer with respect to the workload, but which are not truly ICs (that is, the database designer has no reason to specify these as rules), then these could be expressed as SCs.

So far, we have only discussed SCs that are valid with respect to the current state of the database, in the same sense that ICs are. We can also consider SCs that are "soft" in a second way as well: that the SC statement does not completely hold with respect to the database. Instead, some "violations" of the SC statement are acceptable. If the SC holds for most data, then this could still be useful information. Thus, SCs can be divided into:

- *absolute soft constraints* (ASCs), which have no violations in the current state of the database; and
- *statistical soft constraints* (SSCs), which can have some degree of violation.

For a SSC, we keep two components: the constraint statement itself, and statistical information, such as a *confidence* factor, which is a measure of how closely the data adheres to the constraint. (A SSC with a confidence of 100% is essentially an ASC.) SSC candidates greatly outnumber ASC candidates. Therefore, it may be easier to discover useful SSCs. SSCs are also easier to maintain, since it does not matter when they go slightly stale. Like informational constraints, SSCs do not have to be checked at update. Periodically, the statistics of SSCs should be brought up to date, just as other catalog statistics should be.

The paper is organized as follows. Related work is described in Section 2. In Section 3 we discuss SSCs generally. In Section 4, we discuss issues and strategies for implementing ASCs. Section 5 contains a description of our implementation of the SSCs in DB2 and their use for cardinality estimation. We conclude in Section 6.

2. RELATED WORK

Soft constraints, as described above, have also been called *state constraints* or *temporary constraints* in the literature [9].[1] So the basic idea is not new. Unfortunately, any name in using the word "constraint" is a misnomer; soft constraints do not *constrain* anything![2] Instead, they are characterizations of the database.

Sybase supports a limited class of soft constraints. Sybase will maintain max and min information for a table attribute. This information is available as "constraint" information

[1] Although both of these terms, *state constraints* and *temporary constraints*, have been overloaded in the literature to mean other things too.

[2] In further work, we hope to devise a better name for these.

to the optimizer which can abbreviate range conditions in a query using the SC information. The "SCs" are maintained *synchronously*—that is, at transaction time—so serve as ASCs. On insert, this is clearly no more expensive than checking a check constraint.

In [6], two techniques, *join elimination* and *predicate introduction*, from the *semantic query optimization* literature [4] are refined and implemented in DB2. The paper focuses on join elimination of joins over foreign keys (referential integrity). Introduction of a predicate to a query may offer new access paths via an index which were unavailable before. Rules for selecting a specific subclass of predicates that can be safely introduced are devised. Since the rewrite engine is heuristic based and only one query (rewrite) can be passed to the cost-based optimizer, it must be ensured that these new rewrites will virtually always result in a query that can be more efficiently executed.

They show a marked improvement in performance over standard TPC-D and APB-1 OLAP benchmark queries, and demonstrate that the techniques do not degrade performance elsewhere. Another advantage of the work is that it employs classes of integrity constraints that are commonly defined (referential integrity and check constraints). The experimental results presented employ ICs defined in the benchmarks.

Of course, in environments where such ICs do characterize the data but are not defined as ICs, these techniques cannot work. However, any facility to discover referential integrity and check constraints and to maintain these as SCs would enable these optimization techniques.

Next, we review work to discover specific classes of "SCs" over a database for the purpose of optimization. It is unlikely that many of these SCs would be specified as ICs, so discovery is the only way they would be obtained. It is also unlikely that a DBA would want to promote any of these as ICs since their integrity is not important to the semantics of the database.

In [10], the goal is to discover certain linear correlations between two numeric attributes in the database. This work focuses on searching over pairs of attributes, A and B, which appear together in a given table (call it **table**) and which appear together commonly in workload queries, to discover pairs which are linearly correlated. The goal is to discover a best linear formula of the form

$$\mathsf{A} = k(\mathsf{B}) + b$$

and an ϵ deviation such that all A's values fall within ϵ of $(k(\mathsf{B}) + b)$. This could be expressed by the predicate

$$\mathsf{A} \text{ between } k(\mathsf{B}) + b - \epsilon \text{ and } k(\mathsf{B}) + b + \epsilon$$

Furthermore, this formula should be fairly selective (that is, ϵ is small). So given a value for A, a small range over all the possible B's is selected. To limit the scope of discovery, a threshold is used as a bound for acceptable values for ϵ.

Assume that such a k, b, and ϵ have been discovered for pair A and B with respect to **table**. Given the SQL query

```
select *
    from table
    where B = X;
```

for some constant X, and that there exists an index on **table**.A but no index on **table**.B. The query can be rewritten as

```
select *
    from table
    where B = X and
        A between kX + b − ε and kX + b + ε;
```

This allows for the potential use of the index on A.

In [10], it was decided to explore attribute pairs only within the same table because then discovered correlations could be represented as check constraints. Since DB2 does not support yet inter-table check constraints (no inter-table constraints beyond foreign key), a linear correlation equation between attributes from different tables could not be represented in DB2 as an IC. Of course, it would be possible in principle to mine for these linear correlations between attributes across common join paths. Such information could lead to good optimization possibilities. But we would need a way to represent the correlation information and to make it available to the optimizer.

In [8], the focus is in fact on correlating two attributes, say A and B, from two different tables, say **one** and **two**, respectively, such that **one** $\bowtie_{\mathsf{C}}$ **two** is a common join path. This time, however, the focus is on how to find the "holes" over **one** $\bowtie_{\mathsf{C}}$ **two** with respect to ranges of A and B. That is, selection conditions when A is selected over a given range while B is selected over a given range, no tuples result in **one** $\bowtie_{\mathsf{C}}$ **two**. If one knows these two dimensional "holes", one can trim range conditions on A and B in queries that involve **one** $\bowtie_{\mathsf{C}}$ **two**. This can reduce the number of pages that need to be scanned for the join.

An algorithm is presented in [8] which can discover all the maximal "holes" with respect to a pair of attributes and a join path. The discovery algorithm is quite efficient and is linear in the size of the resulting join table. In experiments, good optimization has been demonstrated through range restriction using the holes.

Extracting semantic information from database schemas and contents, often called rule discovery, has been studied over the last several years. Rules can be inferred from integrity constraints [2, 3, 30] or can be discovered from database content using machine learning or data mining approaches [5, 7, 12, 27, 28, 30]. It has also been suggested that such rules be used for query optimization [13, 27, 28, 30] in a similar way that traditional integrity constraints are used in semantic query optimization [4, 6, 17].

A lot of work has been devoted to the problem of estimating the size of the result of a query expression. Approaches based on sampling were explored in [11, 18] and on histograms in [15, 25]. [21] provides a survey of several tech-

niques and [16] provides an analysis of error propagation in size estimation. Although the information about keys is often used in query result estimates, we are not aware of the use of SCs for that purpose.

It has been observed that functional dependencies (FDs), beyond key information, when explicitly represented can be used for optimization in various cases [29]. They are most effective to optimize group by and order by queries when it can be inferred that some of the group by / order by attributes are superfluous. This can save on sorting costs and sometimes eliminate sorting from the query plan completely.

Since many databases are denormalized, in practice, for supposed performance, or were never normalized in design, FD information can be quite useful in further ways for an optimizer. If so, FD information (beyond keys) could be explicitly represented when known. The opportunity to discover FDs in databases should also generally be good. There has been a fair amount of work in recent years that has improved data mining for FDs [1, 14, 19, 20, 22, 26]. With a good FD mining tool, FD information could be made available as SCs.

3. SOFT CONSTRAINTS

3.1 Application

In DB2, there are three ways in which soft constraints could be used in query optimization and execution.

- In *rewrite*. Currently, certain types of integrity constraints are used in the rewrite engine [6]. More types will be accommodated over time. Any ASCs of the right pattern can likewise be used.
- *Query plan parameterization*. Some leeway can be built into query plans. ASCs can be useful in setting those parameters at run-time.
- *Cardinality Estimation*. SCs can be used in the cost model potentially to obtain better estimates on selectivity factors. ASCs can be used for this, but SSCs could be as well and perhaps more effectively (since SSCs would presumably make available additional statistical information).

Later, we discuss this third application in more detail.

3.2 The SC Process

The work discussed in Section 2, and advances in data mining in general, suggest that semantic characterizations that are useful for optimization can be discovered and exploited. Currently, there is no mechanism in RDBMSs to represent such characterizations (SCs) and to maintain them. Our motivation with SCs is to devise a useful representation for these characterizations that would expedite both their use in the optimizer and their maintenance.

There are three stages in a SC facility:

1. discovery,
2. selection, and
3. maintenance.

Discovery can be done by off-line algorithms or opportunistically during light-load periods for the system. For the linear correlation work [10] and the join hole work [8], there are algorithms for discovery. For FDs, reasonable algorithms may be close at hand.

Each discovery procedure is focused on a particular class of SCs. Furthermore, input from the optimizer, the database's statistics, and the workload can likely be used to direct the search towards those characterization that would be most beneficial. Even so, certain discovered SCs will be more worthwhile than others, and only some will be in fact useful. The selection stage chooses the most promising of the discovered SCs to keep. Which to choose will be based on the estimated utility of each for the optimizer with respect to the optimizer's capabilities, the database's statistics, and the workload. The selection stage could be dynamic, selecting a larger set of SCs to begin and dropping ones that do not prove to be useful over time. SCs might be inexpensively maintained—perhaps not absolutely or maintained asynchronously—but not employed over a probationary period to assess their likely utility.

Any SCs that are chosen must be maintained. The expense of a SC's maintenance must be weighed against its utility. The maintenance cost of certain classes of SCs will be inexpensive, while others' may be prohibitive. Much research is needed on maintenance strategies.

3.3 Implementation: ASCs versus SSCs

ASCs are easiest to understand in a way: Any statement that can serve as an IC potentially can serve as a ASC. Since we know how to write and maintain (verify) many types of ICs, we know as well for those types treated as ASCs.

Much of the optimization work discussed above would need ASCs. Known linear correlations between attribute pairs can be exploited to add a predicate to a query to allow for an index use. This rewrite has to be semantically equivalent to the original query for this query plan to evaluate to the correct answers. Thus only a k, b, and ϵ that classified at 100% (guaranteed) can be used. Likewise, join holes require semantically equivalent rewrites. FDs used to abbreviate group by and order by clauses also must be semantically equivalent for the answer set to be correct.

However, ASCs then are as expensive to maintain as ICs. They must be checked on updates as are ICs. Therefore as with ICs, it would only be practical to support a small number of them for a given database.

We are able to augment DB2 to store ASCs easily. They are treated as flagged ICs. (We have this mechanism already in place in DB2 for informational constraints.) How to maintain them is not as clear. The maintenance policy of last resort is to drop an ASC if it is ever violated. (A modified version might be reinstated later by the discovery engine.) There are many implementation issues about how this could be done efficiently. How to weigh an ASC's potential benefit versus its predicted maintenance cost is also a question.

SSCs allow for greater, and far less expensive, maintenance strategies. Since a SSC reflects the data to some partial degree, absolute currency is not required. SSCs can be maintained asynchronously. Many more "useful" SSCs are likely to be discovered than ASCs simply because SSCs allow for a margin of error.

Meanwhile, SSCs seem to be only useful for filter factoring. (Still, this alone may be quite useful.) There are also general questions as to what form SSCs should have. What statistical information should be kept with SSCs? How much statistical information? For instance, consider an ASC version of linear correlations as discussed above. We might have that 90% of tuples in a table abide the linear equation relating A and B with parameters k, b, and $\epsilon_{90\%}$. Should the databas also keep $\epsilon_{70\%}$ and $\epsilon_{80\%}$?

It is possible that one would want to associate a second measure with ASCs: a measure of *currency*. This is a second dimension of statistics to measure the potential error in the SSC statement, based upon activity since the last time it was updated. Given a fact table of a million records and the knowledge that only a thousand tuples are affected by updates daily, the margin of error for an SSC as a row check constraint on that table will be quite small over the course of several days. But within a month's time, the margin of error would be 3%.

SSCs offer an opportunity to bridge statistical metadata kept for databases and the structural and constraint metadata (that is, ICs) that is kept for them. There are many questions as how to combine best these types of information. We are also interested in what classes of SSCs are most useful for filter factoring.

4. ABSOLUTE SOFT CONSTRAINTS

An ASC should be

- demonstratably useful by the optimizer with respect to the workload,
- not too expensive to check and/or to repair, and
- unlikely to be violated on update.

We have evidence that such ASCs exist for realistic databases and workloads, as discussed in Section 2.

4.1 When an ASC is Violated

We need to resolve whether, or when, an ASC can be and should be used by rewrite. If ASCs are used in query plans, those query plans are in jeopardy when the ASC is violated. Even if the ASC is "repaired", the old query plans are no longer valid. This can have consequences for concurrency and serializability.

There can be problems at run-time due to serializabilty: a transaction (A) that executes a query rewritten by an ASC runs concurrently with another transaction (B) that violated (and so overturns) the same ASC. The possibility exists that wrong answers could be generated by the transaction A.

Dropping an ASC that has become invalid would require an exclusive lock on the catalog. This would have the effect of strictly serializing this transaction with all other sections that depended on that ASC. It would not be good for any I/U/D (insert/update/delete) statement to block in this manner.

Except for a transaction level that allows dirty reads, it does not seem that a transaction conflict with an ASC can arise; by isolation, each transaction sees its own consistent snapshot of the database. However, not every serial equivalence of B then A is acceptable. The issue of serializabilty for ASCs in rewrite needs to be considered and settled.

One possible solution, if there are serializabilty problems and if we want ASCs for rewrite, is as follows. Abort transaction A above since it conflicts with transaction B (in that B overturns an ASC that A employs). Re-issue transaction A (modified now not to use the ASC) after B commits. The re-issue can be done behind the scenes just as is done in the case of deadlock resolution. So the user who issued A sees no difference except for more wait time. If our ASCs are such that a violation event (like transaction B) is rare, this will not happen often. Of course, what then if transaction A aborts in the end anyway? Is the ASC then re-instated?

For reasons of practicality, deadlock resolution is done at the application layer today. This means that the application for transaction A would have to re-issue. We would need to be careful when choosing to abort: transaction B may have just started while transaction A has been running for hours. We are pursuing how to resolve these issues.

A worse expense for ASC violations is that every pre-compiled query plan that employes a violated ASC in its plan must be dropped. This can be a serious issue for the workload, since those queries must be recompiled before they can be used again. Again, if ASC violation is a very rare event, this may be acceptable. This deserves study. One possilbe tactic is for a package to incorporate a "backup" plan which is ASC-free. If an ASC is overturned, a flag is raised and packages revert to the alternative plans.

Of course, for workload environments that predominantely consist of dynamic queries (write and run once), this is not an issue. Data warehousing environments are like this. Also, it would be possible to restrict the use of ASCs in rewrite just to dynamic queries and never for precompilation.

It may also be possible to cirumvent these issues altogether. We may be able to support the notion of an *absolute* SC but allow for exceptions all the same. This would require some type of "exception handling" be built into any query plan that employs an ASC. Then whenever the ASC is violated, none of these inherent problems arise. We discuss such a possibility in Section 4.4 in which we treat ASCs as a type of AST. It may be possible to devise other ways to circumvent these problems as well.

4.2 Runtime Optimization

It may be worth considering ASCs just for runtime query parameterization. No ASCs would be used in rewrite. Significant changes would be needed in the optimizer in order

to consider the available ASCs and to add the extra parameters. Certain classes of ASCs would be amenable to this approach. Sybase's min and max "ASCs" are used effectively in this way. The actual values in the ASC are not important, such as what the actual maximum value is. Rather, the availability of this information (of the ASC) at runtime is important. This does mean that the ASC has to be available whenever the query is executed. So the query plan is reliant on the ASC in that respect. Thus, ASCs used in this way would have to be maintainable.

4.3 Maintenance

It may be that for certain classes of ASCs incorporation of the ASCs in the query plan will never change the answer set regardless. If the ASC is in violation, the query plan may just be sub-optimal. It would be interesting to see whether there exist such classes and whether any are useful for the optimizer. Maintenance could be much less expensive for these.

When an ASC is violated, there are different maintenance strategies. The simplest is to drop the ASC. Another possibility may be to attempt to "repair" the ASC. The repair could either be done synchronously (at the time of the violation) or asynchronously (dropped from active, and queued for repair).

While ASCs require some synchronous maintenance to assure that the ASC remains consistent with respect to the database (or is dropped), hybrid approaches of synchronous and asynchronous repair are possible. A less expensive synchronous repair that is suboptimal can be performed. Then an asynchronous repair can return the ASCs to optimal characterization. For example, consider join holes. Because the ASC is inter-table, any absolute maintenance requires a join. Consider an insert to table **one** which adds a new tuple with a new value for A. Furthermore, consider that the new value for A occurs in holes with respect to B. To see whether these holes are now violated requires a join of the new tuple in **one** with table **two**. However, one can just assume that the new value for A does violate the holes without checking. The holes can be dropped (or split). This is suboptimal since there may have been no violation in fact. The hole ASCs would be brought back to optimality the next time the asynchronous join hole algorithm is run.

4.4 ASCs as ASTs

Both ASCs and SSCs are closely related to materialized views, and thus the *automated summary table* (AST) facility in DB2.

Just as it has been recognized that IC checking can be considered as sub-problem of materialized view maintenance, any IC can be rethought of as a materialized view that must always be empty. Thus materialized view maintenance can be used for ASC checking is as well, and ASCs could be maintained as materialized views (ASTs).

A way to handle ASCs may be as follows. An IC can be considered as a (materialized) view that is always empty. It may not be empty, in which case the materialized view explicitly represents the exceptions to the ASC. This differs still conceptually from a SSC since we are not keeping any aggregate or statistical information. Thus updates that violate the ASC are allowed. The exceptions are just stored. The assumption is that for a reasonable ASC, the materialized view that represents it is always nearly empty.

Now any query plan that employs an ASC must also process the exceptions. In cases that the ASC's AST is empty, the exception addendum to the query plan should be of trivial cost. Consider a table **purchase** and the attributes order_date and ship_date. We can have the SSCs available as discussed above. For example, for 99% of tuples, the ship_date is between the order_date and three weeks later. But now we could support the business rule that products are shipped within three weeks directly.

```
create summary table late_shipments * as
    (select *
        from purchase
        where ship_date > order_date + 3 weeks);
```

The ASC representing the business rule states that there should not be shipments later than three weeks. The AST **late_shipments** above tracks the exceptions (about 1% of the tuples).

Consider the query

```
select *
    from purchase
    where ship_date = '15 Dec 1999';
```

Consider that there is an index on order_date, but not on ship_date. We would like to introduce a predicate with order_date then to allow a better access path. From the SSC information, we know that 99% of the answers (on average) would be captured by the query

```
select *
    from purchase
    where ship_date = '15 Dec 1999' and
          order_date ≥ '15 Dec 1999' − 3 weeks;
```

Since we have the AST **late_shipments** available, we can capture the remaining 1%:

```
(select *
    from purchase
    where ship_date = '15 Dec 1999' and
          order_date ≥ '15 Dec 1999' − 3 weeks)
union all
(select *
    from late_shipments
    where ship_date = '15 Dec 1999');
```

If the second sub-query is reasonably inexpensive, this can result in a better query plan. Note that we can use union all regardless since we know that the two sub-queries must return mutually distinct tuples anyway. This means that the optimizer has more options when optimizing the query.

Consider again the AST **late_shipments** defined above. This AST can be used to influence filter factor estimation. The table **late_shipments** can be defined in DB2 v7. DB2's

current AST facility allows one to define effectively a materialized view via a single table select statement without use of aggregation.

Such ASTs are used already by the optimizer. Any AST that matches over a query's predicates is appended as a choice-point in the query plan. This query plan is passed to cost-based optimizer. The cost-based optimizer may decide to route through the AST instead of the base tables (or not) depending on cost estimates. In either case, the optimizer uses the statistics from *both* the base tables and the ASTs involved for filter factor estimation for a better estimate. Thus the existence of the AST **late_shipments** could improve filter factor estimation for queries over the base table **purchase**.

In concurrent work, the idea of *information* AST*s* is being developed by the DB2 group. An information AST is not materialized, but runstat information is maintained for it. So it is not routable, but can be used for filter factor estimation. So if we did not need to keep the AST **late_shipments** materialized (say we are not using ASTs as ASCs for rewrite), this could be kept as an information AST.

This offers an immediate means to convey certain types of "SSC" information as well to the optimizer. These SSCs can be represented as ASTs, which then are employed by the optimizer effectively for filter factor estimation.

We are exploring ways that the optimizer can exploit AST information. In the scenerio above, it really is only the runstats information about the AST which is pertinent. However, it seems more natural that the *data* of the AST itself will represent statistical profiling information about other tables, joint distributions for attributes, and join path profiles. Consider the types of histogram queries we are able to write with scalar aggregate operators. Assume that we were able to declare such queries as ASTs.[3] The runstats information about these ASTs *themselves* is irrelevant. It is the data itself (serving as meta-data) that is relevant. How the optimizer can be told when extra profiling information is available via an AST, and how the optimizer can efficiently exploit the ASTs, is a good question to address.

5. STATISTICAL SOFT CONSTRAINTS

In this section, we discuss the application of SSCs in better cardinality estimation.

The estimated cost of accessing the data at various stages in a query plan is determined by a number of factors, such as the disk data transfer rate, the processor speed, the memory or buffers available and the statistics associated with the data. These are all part of a complex model used to generate the final access plan. In order to determine the cost of the various operations involved in gathering and processing the data, one of the key factors is the number of rows, or the cardinality, of the intermediate results. This is used to compute the cost of the subsequent operations. Commercial database systems like DB2 keep various statistics of the data within columns for use in cardinality estimation. These include the number of distinct values, high and low values, frequency and histogram statistics. The optimizer uses these statistics when considering the various predicates within the query to estimate the selectivity that results when these predicates are applied.

Exploitation of constraints for internal rewrites has been suggested before [6]. An example of such exploitation in the query rewrite engine is their use in a union all view, where each branch of the union contains a range of values pertaining to a given column. These ranges are the constraints associated with each branch of the union all view. For example, the first branch contains data corresponding to January; the second contains data corresponding to February and so on for 12 months. A query, with a predicate asking for data from January to March against this union all view, requires us to only look at the first three branches of the union all view. The way this is done is to match the query predicates against predicates in each branch that are introduced according to the corresponding branch constraint. This matching can then be used to knock off the branches of the union view that we know will not contain any data that will satisfy the query. The key requirement here is that the constraints are valid and if actually applied as predicates, they do not change the results of the query.

5.1 Cost-Based Optimization

We extend this concept of introducing constraint-based predicates further to help estimate cardinalities within the optimizer. The main difference is that unlike with the exploitation in the query rewrite engine, the generated predicates are not actually applied. We mark these predicates as special predicates for use in the optimizer only. This allows us to make use of constraints that are not necessarily valid for all the data. They belong to the class of SSCs. It is sufficient that they are true for a certain percentage of the data. Given this additional information of how many rows satisfy the constraint, we could alter the estimated cardinalities. There are two suggested mechanisms of specifying the constraints. The first is to use the same infrastructure as a regular constraint along with an additional number that specifies the percentage of the number of rows that satisfies the constraint. The second is to combine multiple SSCs in virtual columns where the distribution statistics on the virtual column can be broken down into the individual SSCs. These statistics characterize the data in a way that they can be made to appear like predicates in the query within the optimizer. They are marked as such for use by the optimizer for the purpose of cardinality estimation only.

A major issue when estimating cardinality is when one or more predicates involve multiple columns in a table making it difficult to estimate cardinality because of statistical correlation between values in the columns. As an example, consider a **project** table and a query to find the number of projects active on a given day. The predicate that could be used is:

start_date <= '1999-11-15' and end_date >= '1999-11-15'.

It is difficult for the optimizer to estimate the cardinality for the combined effect of these predicates without knowledge of the statistics of the spread between start_date and end_date.

[3]DB2 does not currently support scalar aggregates in ASTs.

A technique used today is to simply treat the columns independently and multiply the filter factors of individual predicates. If '1995-11-15' is somewhere in the middle of the column values that spanned 10 years of data, each predicate would have a selectivity estimate of about 50%. This would work out to a combined estimate of 25% of all the projects done in 10 years! One can see that, if projects rarely went over one month, we should on average have rows corresponding to projects that were done in, say, November 1995 give or take a few that started in October and some that ended in December.

5.2 Twinning

Consider the case when we keep a SSC in the form end_date − start_date <= 30 (assume that this holds for 90% of the rows). We also keep this '90%' number associated with the constraint. For the purpose of cardinality estimation, even if we assume the most likely maximum end_date = start_date + 30, we could add a new predicate that replaces the end_date in the original predicate with one using start_date + 30. Now we have a new predicate start_date + 30 >= '1999-11-15' that we *twin* to our original predicate end_date >= '1999-11-15'. The twinning mechanism simply tells the optimizer that it could use either the original predicate or the new predicate for the purpose of cardinality estimation. Today, DB2 does use a twinning mechanism to allow the optimizer to choose between alternate forms of predicates. For example, an IN predicate could be written as a set of OR predicates. The main difference with the proposed twinning mechanism is that the twinned predicates are for cardinality estimation only. We now have two predicates on the start_date column, the original predicate on start_date and the newly added predicate on start_date that is twinned to the end_date predicate. We can conveniently use these to get better cardinality estimates. This is essentially reducing the range predicates on two columns to a pair of range predicates (or a between predicate) on a single column. The 90% statistic gives us a confidence factor that we may use to adjust the estimate. This may be done to apply upper and lower bounds on our estimates. Assume we calculated a filter factor of F and we have a 90% confidence factor. We should be able to apply a statistical adjustment based on this confidence factor in order to arrive at a better bound than we get using the independence assumption.

In another example, consider finding the number of projects completed in 5 days. The predicate used in the query could be end_date − start_date <= 5. In the previous example, we had the SSC:

end_date − start_date <= 30 [90%].

This is not particularly useful for the cardinality estimation of <= 5 days predicate. It would be desirable to keep this SSC as well, although it does not apply to as many rows as the <=30 day predicate. In this case, say, we have 20% of the rows satisfying the constraint end_date − start_date <= 5. We now have a good estimate for the predicate within the query. Given a number of these SSCs varying the constants (number of days in our example) and the associated percentages with each of the SSCs, the question becomes how to decide which constraint to use. It would seem that a suitable method for numeric data is to use interpolation. This is not very different from the use of histogram statistics within databases today.

5.3 Virtual Columns

The technique suggested is to combine all similar SSCs and group them together as a single entity that provides information for several predicates involving start_date and end_date. For situations similar to the example above, the expressions involved in the SSCs are normalized so that we have an expression on the left-hand side and a constant on the right hand side. In other words, if the constraint was expressed as end_date = start_date + 10, we normalize it to end_date − start_date = 10. This becomes our virtual column expression. The statistics for the set of SSCs or the virtual column are gathered either as part of the mining process or explicitly using SQL statements that run through the data (possibly sampled). Alternatively, actually materializing the expressions into a column in a table and using the traditional method to collect statistics is a brute force approach. The results are compiled in a similar way that the statistics collection routine does for a single column. The advantage of the virtual column is that we do not have the overhead of storing and maintain the materialized data. Exploiting an actual generated or derived columns statistics could be done using similar techniques described here.

Using the virtual column VC (end_date − start_date), we collect frequency and histogram statistics. By looking at the 2 most frequent values of 20 and 30, for example, we would be able to infer that end_date − start_date = 30 for 30% of the rows, end_date − start_date = 20 for 15% of the rows and so on. Histogram statistics might look like:

VC Value X	rows satisfying VC <= X
5 days	1000
10 days	3000
20 days	20000
30 days	80000
50 days	95000

This provides us with condensed information that would otherwise require us to store several SSCs in the traditional form along with associated confidence factors. The following SSC can be generated internally (assuming 100,000 rows):

end_date − start_date <= 5 [1%]
end_date − start_date <= 10 [3%]
end_date − start_date <= 20 [10%]
end_date − start_date <= 30 [80%]
end_date − start_date <= 50 [95%]

We can now use these to generate suitable twinned predicates that allow us to provide bounds to the filter factor estimates used for cardinality estimation.

So far our examples describe techniques more suitable for predicates involving numeric or date predicates involving arithmetic operations.

A simple application of using virtual column or generated column statistics that does not necessarily involve multiple columns might be to get better cardinality estimates for columns that are wrapped by functions. A predicate such as

uppercase(name) = SMITH would make it impossible to estimate if we do not know what the function uppercase does. An SSC that told us that uppercase(name) = SMITH is valid for 15% of the rows would be invaluable. If we had various such SSCs compiled into a virtual column that hinted at the estimate for uppercase(name) = SMITH, uppercase(name) <= AAA, uppercase(name) <= BBB, etc., we would be able to do a much better job when similar predicates appeared in a query. For example, selectivity estimates for the predicates uppercase(name) = SMITH or uppercase(name) like A% would be fairly straightforward. Note that the like predicate in this situation can be split into range predicates using the high and low collating sequence characters to specify the bounds. Note also that range predicates on character columns can use suitable conversion to floating point numbers to do interpolation. Another thing to note is, that in some situations, after breaking up the virtual columns statistics into potential predicates, we may be able to directly set the selectivity of the original predicate without twinning any additional predicates. This can be done for simple situations where there is a one to one mapping of the original predicate and the predicate based on the virtual column or generated column.

5.4 Complex Conditions

Often, correlation between values in more than one column is not very straightforward. An example of a simple CASE based or conditional constraint is provided here to illustrate how we could fail to detect correlation with the independence assumption. Later on, we will provide a possible solution using one or more virtual columns. Consider the constraint:

CASE
  WHEN $c1 = A$ THEN
    $f1(c2)$ between 1 and 100
  ELSE
  WHEN $c1 = B$ THEN
    $f2(c2) >= 1000$
  ELSE
    $c2 = 0$
END.

Here $f1$ and $f2$ are functions to illustrate the more complex case instead of simple columns. The assumption here is that the conditions used within the CASE expression do not overlap. For simple *column op constant*, maintaining multi-column statistics as some databases do might allow us to exploit similar techniques to use for cardinality estimation. Let us assume that the cardinality estimate is 50% for all the rows with $c1 = A$. Without knowledge of the correlation information extracted from the CASE expression, we would not be able to give a good cardinality estimate for a predicate like $c1 = A$ and $f1(c2)$ between 1 and 80. Without analyzing the check constraint information, we would treat the columns independently. The filter of the $c1 = A$ predicate would be 0.5 factor (using probabilities instead of a percentage). That of $f1(c2)$ between 1 and 80 might not be so easily obtained since $f1$ may not be easily interpreted by the optimizer. Typically a default filter factor is used today. Even if we did get it correct (say 0.4 with uniform distribution of $f1(c2)$), our combined filter factor estimation based on independence would be 0.5 * 0.4 = 0.2. One can see that, assuming uniform distribution of the values of $f1(c2)$ between 1 and 100, the filter factor should be closer to 0.4.

Extending this further, if the query contained the predicates $c1 = A$ and $f1(T1.c2) = T2.c2$, we might be able to get a better estimate on the cardinality of the join predicate if we could twin a predicate $f1(T1.c2)$ between 1 and 80. If this were an IC, we could actually introduce and use this predicate to our advantage. For the purpose of cardinality estimation, this could make a very significant difference since data is often very skewed. Knowing that there is a functional dependency alone is not enough. The relation of the values is important for cardinality estimation. In many data warehouse applications, a common design is to have application dimension keys tied to surrogate keys that are compact and used to link a fact table with the dimension table. The queries themselves have the filtering predicates on the application keys. It is extremely useful to apply the technique above and add a predicate $T2.c2$ between 1 and 80 through transitivity on the fact table. This information would allow us to better estimate the cardinality accounting for the skew in the fact table.

While the set of conditions in the constraint could be arbitrarily complex, we could keep statistics for the simpler situations that the optimizer could exploit. In the example above, we might want to keep the statistics for the two branches of the CASE statement as separate virtual column entities using the expressions c1 || $f1(c2)$ and $c1||f2(c2)$ with appropriate values of $c1$(A or B). Note that || is not true type compatible concatenation that we require in SQL but a *pseudo-concatenation*. This is more a representation of the values of the two columns placed side by side when gathering statistics similar to the way we would do when collecting statistics in a multi-column index. With the conditional predicates split into separate virtual column expressions based on the WHEN clause conditions in the CASE expression, it is less complex to analyze during query rewrite.

The histogram statistics for the virtual column containing $c1 = A$ will look like :

$c1\|\|f1$	Number of rows
A || 10	100
A || 20	200
.....	...
A || 100	1000

Given a predicate $c1 = A$ and $f1(c2)$ betweeen 1 and 75 which is rewritten internally as where $c1 = A$ and $f1(c2) >=$ 1 and $f1(c2) <= 75$, we could look for the virtual column containing $c1 = A$ and then twin the predicate $c1||f1(c2) >= A||1$ and $c1||f1(c2) <= A||75$ to the combined original predicates.

The twinning mechanism needs to suitably tag the twinned predicate or predicate set with the original predicate or predicate set. In the current version of DB2, there is a mechanism to adjust for correlation based on information within the indexes. This mechanism makes use of a correlation adjustment vector that contains sets of predicates and corresponding adjustment factors. When all the predicates ap-

pear together for the first time in an access plan, we apply the adjustment for the set making sure that we roll out adjustments that might have been applied for any subset. This same mechanism might be exploited to adjust for correlation when adjustments are made to original predicates based on the information using the twinned predicates on the virtual columns. The original predicate set would be entered into the correlation adjustment vector and the adjustment factor would be computed based on the selectivity estimates of the twinned set and the original set.

To start with, the set of statistics similar to that obtained by explicitly collecting statistics on the materialized values of the column could be used. The suggested method is to use an extensible approach amenable to adding new statistics if desirable. To make this extensible, these statistics are packaged in a form that can be suitably stored in the catalogs. One method is to compose an XML structure that has the relevant statistical information. When the query is compiled, the information is interpreted (converted from XML to internally recognizable formats) and attached to the table as a virtual column statistical attachment in a manner similar to the way indexes are associated with the table. Note that we could also identify the constraint attachments if they were real integrity constraints. In the above examples, given a generated column it may be possible to exploit the generated predicates during execution and derive the benefits of better index usage either on the underlying column or on the generated column if either the original predicate or that on the generated column were used. The cardinality estimate would still be governed by the predicate on the generated column.

To summarize, given integrity constraints, some databases like DB2 have the ability to do some semantic query optimization during the query rewrite phase. An additional task during this phase involves further analyzing predicates within the query with each of the statistical constraints derived from the virtual column statistical attachment. Since these may not be valid integrity constraints, we cannot do semantic query optimization. We can exploit this information in the optimizer to better estimate cardinalities. In simple situations, if we find an exact expression match between the predicate in the query and that of the partial constraint, we simply mark the original query predicate with the selectivity for the optimizer to use. In more complex situations, with multiple predicates involved in the query and the query predicates do not match the partial statistical constraint predicates exactly (other than involving the same columns), we are be able to add *twin* predicates to the original query predicates based on equivalence. In general twinning predicates is a technique to provide alternate predicate forms where either predicate could be used at execution time. In the case of SSCs, however, the generated predicates are for cardinality estimation only and are not to be evaluated during execution of the plan. The twin predicate could be used by itself for cardinality estimation of the original predicate or, alternatively, a combination of the twin predicates (and possibly original predicates) in the query may be more amenable to account for correlation between values in the columns.

6. CONCLUSIONS

Informational constraints and their variants are not a new idea, and they have now been implemented by several commercial systems. Such techniques have been recognized as critical in order to allow for reduced cost of constraint processing. Soft constraints are new, to the best of our knowledge. They offer an additional mechanism to provide useful constraint-like information to the optimizer while not improperly overloading the notion of integrity constraints. Statistical soft constraints offer a way to capture complex statistical information about the database in a declarative manner similar to integrity constraints in a succinct form. Statistical characterizations that would be impossible to represent by traditional statistical profiling (such as with runstats) can be captured this way. Furthermore, basic statistics handling in database systems are "hardwired". An optimizer equipped to handle SSCs can be trained for better cost estimation with the addition of SSCs.

7. ACKNOWLEDGMENTS

We would like to thank Bill O'Connell, Haoping Xu, and Qingsong Yao for their help with this paper.

8. REFERENCES

[1] D. Bitton, J. Millman, and S. Torgersen. A feasibility and performance study of dependency inference. In *Proceedings of the Fifth ICDE*, pages 635–641, Los Angeles, California, USA, 1989. IEEE Computer Society.

[2] S. Ceri, P. Fraternali, S. Paraboschi, and L. Tanca. Automatic generation of production rules for integrity maintenance. *TODS*, 19(3):367–422, 1994.

[3] S. Ceri and J. Widom. Deriving production rules for constraint maintanance. In *Proceedings of the* 16^{th} *VLDB*, pages 577–589, Brisbane, Australia, 1990.

[4] U. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM TODS*, 15(2):162–207, June 1990.

[5] I.-M. A. Chen and R. C. Lee. An approach to deriving object hierarchies from database schema and contents. In *Proceedings of the 6th ISMIS*, pages 112–121, Charlotte, NC, 1991.

[6] Q. Cheng, J. Gryz, F. Koo, C. Leung, L. Liu, X. Qian, and B. Schiefer. Implementation of two semantic query optimization techniques in DB2 UDB. In *Proc. of the 25th VLDB*, pages 687–698, Edinburgh, Scotland, 1999.

[7] W. Chu, R. C. Lee, and Q. Chen. Using type inference and induced rules to provide intensional answers. In *Proceedings of the 7th ICDE*, pages 396–403, Kobe, Japan, 1991.

[8] J. Edmonds, J. Gryz, D. Liang, and R. J. Miller. Mining for empty rectangles in large data sets. In *Proceedings of the 8th ICDT*, pages 174–188, London, UK, 2001.

[9] P. Godfrey, J. Grant, J. Gryz, and J. Minker. Integrity constraints: Semantics and applications. In G. Saake and J. Chomicki, editors, *Logics for Databases and Information Systems*, pages 265–307. Kluwer Academic, 1998.

[10] J. Gryz, B. Schiefer, J. Zheng, and C. Zuzarte. Discovery and application of check constraints in DB2. In *Proceedings of ICDE*, Heidelberg, Germany, 2001.

[11] P. Haas, J. Naughton, S. Seshadri, and L. Stokes. Sampling-based estimation of the number of distinct values of an attribute. In *Proceedings of VLDB*, pages 311–322, Zurich, Switzerland, 1995.

[12] J. Han, Y. Cai, and N. Cercone. Knowledge discovery in databases: An attribute-oriented approach. In *Proceedings of the 18th VLDB*, pages 547–559, Vancouver, Canada, 1992.

[13] C. N. Hsu and C. A. Knoblock. Using inductive learning to generate rules for semantic query optimization. In *Advances in Knowledge Discovery and Data Mining*, pages 425–445. AAAI/MIT Press, 1996.

[14] Y. Huhtala, J. Karkkainen, P. Porkka, and H. Toivonen. Efficient discovery of functional and approximate dependencies using partitions. In *Proceedings of 14th ICDE*, pages 392–401, Orlando, FL, Feb. 1998.

[15] Y. E. Ioannidis. Universality of serial histograms. In *Proceedings of 19th VLDB*, pages 256–267, Dublin, Ireland, 1993.

[16] Y. E. Ioannidis and S. Christodoulakis. Optimal histograms for limiting worst-case error propagation in the size of the join results. *TODS*, 18(4):709–748, 1993.

[17] J. King. Quist: A system for semantic query optimization in relational databases. In *Proc. 7th VLDB*, pages 510–517, Cannes, France, Sept. 1981.

[18] R. Lipton, J. Naughton, and D. Schneider. Practical selectivity estimation through adaptive sampling. In *Proc. of Sigmod*, pages 40–46, 1990.

[19] S. Lopes, J. Petit, and L. Lakhal. Efficient discovery of functional dependencies and armstrong relations. In *Proceedings of the 6th International Conference on Extending Database Technology*, pages 350–364, Konstanz, Germany, 2000. Springer.

[20] H. Mannila and K.-J. Raiha. Algorithms for inferring functional dependencies from relations. *Data and Knowledge Engineering*, 12(1):83–89, 1994.

[21] M. Mannino, P. Chu, and T. Sager. Statistical profile estimation in database. *ACM Computing Surveys*, 20(3):191–221, 1988.

[22] N. Novelli and R. Cicchetti. Fun: An efficient algorithm for mining functional and embedded dependencies. In *Proceedings of the 8th ICDT*, pages 189–203, London, UK, 2001.

[23] *SQL Reference Manual, Oracle 8i, Realease 8.1.5*. 500 Oracle Parkway, Redwood City, CA 94065, 1999.

[24] G. N. Paulley and P.-Å. Larson. Exploiting uniqueness in query optimization. In *Proceedings of the 10th ICDE*, pages 68–79, 1994.

[25] V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita. Improved histograms for selectivity estimation of range predicates. In *Proceedings of SIGMOD*, pages 294–305, Montreal, Canada, 1996.

[26] I. Savnik and P. Flach. Bottom-up induction of functional dependencies from relations. In G. Piatetsky-Shapiro, editor, *Knowledge Discovery in Databases*, pages 284–290. Morgan Kaufman Pub., 1993.

[27] S. Shekar, B. Hamidzadeh, A. Kohli, and M. Coyle. Learning transformation rules for semantic query optimization. *TKDE*, 5(6):950–964, Dec. 1993.

[28] M. Siegel. Automatic rule derivation for semantic query optimization. In *Proceedings of the 2nd International Conference on Expert Database Systems*, pages 371–386, Vienna, Virginia, 1988.

[29] D. Simmen, E. Shekita, and T. Malkems. Fundamental techniques for order optimization. In *Proceedings of SIGMOD*, pages 57–67, 1996.

[30] C. T. Yu and W. Sun. Automatic knowledge acquisition and maintenance for semantic query optimization. *IEEE Transactions on Knowledge and Data Engineering*, 1(3):362–375, Sept. 1989.