open proceedings

Business-Intelligence Queries with Order Dependencies in DB2

Jaroslaw Szlichta
University of Toronto &
IBM Toronto
Centre for Advanced Studies
szlichta@cs.toronto.edu

Parke Godfrey
York University in Toronto &
IBM Toronto
Centre for Advanced Studies
godfrey@cse.yorku.ca

Jarek Gryz
York University in Toronto &
IBM Toronto
Centre for Advanced Studies
jarek@cse.yorku.ca

Wenbin Ma
IBM Toronto Laboratory
wenbinm@ca.ibm.com

Weinan Qiu
IBM Toronto Laboratory
davidqiu@ca.ibm.com

Calisto Zuzarte
IBM Toronto Laboratory
calisto@ca.ibm.com

ABSTRACT

Business-intelligence queries often involve SQL functions and algebraic expressions. There can be clear semantic relationships between a column's values and the values of a function over that column. A common property is *monotonicity*: as the column's values ascend, so do the function's values. This we call an *order dependency* (OD). Queries can be evaluated more efficiently when the query optimizer uses order dependencies. They can be run even faster when the optimizer can also reason over known ODs to infer new ones.

Order dependencies can be declared as *integrity constraints*, and they can be detected automatically for many types of SQL functions and algebraic expressions. We present optimization techniques using ODs for queries that involve *join*, *order by*, *group by*, *partition by*, and *distinct*. Essentially, ODs can further exploit *interesting orders* to eliminate or simplify potentially expensive sorts in the query plan. We evaluate these techniques over our implementation in IBM® DB2® V10 using the TPC-DS® benchmark schema and some IBM customer inspired queries. Our experimental results demonstrate a significant performance gain. We additionally devise an algorithm for testing logical implication for ODs which is polynomial over the size of the set of given ODs. We show that the inference algorithm which we have implemented in DB2 is sound and complete over sets of ODs over *natural domains*. This enables the optimizer to infer useful ODs from known ODs.

1. INTRODUCTION

1.1 Motivation

As business-intelligence (BI) applications have become more complex and data volumes grow, so have the analytic queries needed to support them. This increasing complexity raises performance issues and numerous challenges for query optimization. Worse, traditional optimization methods often fail to apply when logical subtleties in database schemas and in queries circumvent them. For example, data-warehouse schemas will use surrogate keys, while predicates in business analytic queries will use natural values (such as sale_date = '2010-07-01'). Real world queries will use *SQL functions* (such as year(d_date)) and *algebraic expressions* (such as d_date + 30 days).

These subtleties cause the optimizer to miss opportunities to use indexes, partition elimination and pipeline operations, and to add potentially expensive operations such as sort even when the data is already sorted appropriately. This is because *semantic relationships* between the functions and expressions the queries use and the data in the database—and between data themselves in the schema, as between surrogate and natural keys—are opaque. If these relationships could be discovered and used, more efficient query plans would result.

The relationship on which we focus in this work is *order*. If the rows of a table were ordered by its date column d_date, they would also necessarily be ordered by d_date + 30 days. Indeed, the *function* (over d_date) of d_date + 30 days is *monotonically increasing* with respect to d_date. For this, we say d_date *orders* d_date + 30 days.[1] If an index on d_date could be used to provide results ordered by d_date, then the same index would provide the results ordered by d_date + 30 days, since this is the same order. This semantic relationship of order is a type of *dependency*, and we call it an *order dependency* (OD) [17, 18, 21]. It is akin to the well-known concept of *functional dependencies* (FDs). (In fact, ODs strictly subsume FDs.)

While it will be readily obvious to any reader that d_date orders d_date + 30 days, this observation is not for free for the optimizer. It would need explicit mechanisms to recognize the dependency. While this particular order dependency rightfully seems trivial, we shall see there are many that are not. Then "when" and "how" to exploit such dependencies in query planning is far from trivial too. This work is about this aspect of query optimization.

Consider then the SQL query in Query 1 over the TPC-DS[2] schema. In the schema, date_dim is a *dimension* table with the primary key d_date_sk with one row per day. (The attribute d_date_sk is a sequential number.) The table has columns d_date,

[1] In this case, d_date + 30 days *orders* d_date also. We then say the two are *order equivalent*. However, "orders" is not inherently symmetric. Consider year(d_date) and d_date. In this case, d_date orders year(d_date), but year(d_date) does not order d_date.

[2] http://www.tpc.org

10.5441/002/edbt.2014.81

```
select D.d_date + 30 days,
       max(S.ws_ext_sales_price) as most
from date_dim D, web_sales S
where S.ws_sold_date_sk = D.d_date_sk and
      D.d_date between
          date('1998-01-01') and
          date('2002-01-01')
group by D.d_date + 30 days
order by D.d_date + 30 days;
```

Query 1: Plus thirty days.

d_month, d_quarter, and d_day, and additional columns that qualify the day (such as whether it is the weekend, a holiday, and, if so, the name of the holiday). The table web_sales is a large *fact* table recording all individual sales, with ws_sold_date_sk as a foreign key referencing date_dim on d_date_sk.

Let there be a tree index for date_dim on d_date. The optimizer will miss that the index could be used in evaluating Query 1 to accomplish both the group-by and the order-by. How might the query be rewritten manually to resolve this?

- group by d_date + 30 days and order by d_date:
  This is not legal SQL; the attribute in the order-by is not listed in the group-by (as such).
- group by d_date and order by d_date + 30 days:
  This is accepted by DB2; derived attributes—functions and algebraic expressions derived over the attributes listed in the group-by (which may include derived attributes itself)—can be used in the select and order-by clauses.
  However, this does not resolve the inefficiency. The query plan still explicitly sorts to "satisfy" the order-by.
- group by d_date and order by d_date:
  This does work! The index can now be employed to implement the group-by and to satisfy the order-by.

Of course, it is not the responsibility of the SQL programmer to write queries painstakingly—or of an automated BI report system that generates SQL queries in the back-end—in such a way to assure the optimizer will handle it well. This would violate the declarative principle of SQL. Even if we tried to put the onus on programmers to be careful, they cannot be expected to know what is problematic and what is not. While a clever SQL programmer can sometimes skirt such pitfalls by careful composition (as here), more often it is not possible. So, we have to fix it. The optimizer needs to recognize that d_date and d_date + 30 days are semantically equivalent for order, thus skipping the superfluous sorting step, regardless of how the query is written.

Next, consider Query 2. In SQL, *date* and *time* are complex data types. These are central to BI applications, and provide for rich *drill down* and *roll up*. In TPC-DS in table date_dim, some of date's hierarchy is materialized in columns: d_year, d_quarter, d_month, and d_day.

```
select D.d_year, D.d_quarter,
       D.d_month, D.d_day
       sum(S.ws_sales) as total
from date_dim D, web_sales S
where S.ws_date_sk = D.d_date_sk and
      D.d_year between 2001 and 2004
group by D.d_year, D.d_quarter,
         D.d_month, D.d_day
order by D.d_year, D.d_quarter,
         D.d_month, D.d_day;
```

Query 2: Eliminating quarter.

Let there be a tree index for date_dim on d_year, d_month, d_day. Unfortunately, this index would not help in a query plan, even for the group-by: d_quarter intervenes. Note that d_month *functionally determines* d_quarter. The query's author cannot eliminate mention of d_quarter in the group-by, however, as it appears in the select. Fortunately, by the work in [16], DB2 can eliminate it internally from the group-by, based on the recognition of the functional dependency (FD). The index can then be used to implement the group-by operation.

However, this FD, d_month → d_quarter, is *not* logically sufficient likewise to remove d_quarter from the order-by clause. The optimizer must still apply a sort operator to "satisfy" the order-by directive. However, because d_month *orders* d_quarter—which says more than just that d_month *functionally determines* d_quarter—the attribute d_quarter can be removed from the order-by clause also, to result in a semantically equivalent query.[3] In this work, we show how this is accomplished.

1.2 Contributions and Outline

In Section 2, we provide background on order dependencies—notational conventions and definitions—as we use in this paper, and considerations that arise in data-warehouse schema design. In Section 3, we address how to use order dependencies in query optimization. There are two aspects to this: how and where the optimizer makes use of OD information; and how OD information is discovered.

1. *Optimizing with Order Dependencies.*
   (a) Section 3.1 is divided into two sections. In Section 3.1.1, we go into further depth how ODs are used to optimize.
   (b) In Section 3.1.2 we present two inference algorithms and show where and how they are invoked in the optimizer: *Reduce Order OD* which puts ODs into a canonical form for matching against *interesting orders*; and *Homogenize Order OD* which discovers equivalent columns, order-wise. We discuss the utility of these algorithms.
2. *Detecting Order Dependencies.* In Section 3.2, we show how ODs between columns and functions over columns (SQL functions and algebraic expressions) can be automatically detected by the optimizer.
   (a) These techniques have been implemented as a prototype within DB2.
   (b) We present a suite of real-world IBM customer queries over TPC-DS benchmark that illustrate the issues, which are then used in Section 4 for an experimental performance evaluation. The optimizer automatically infers the associated OD information and uses it to produce the improved query plans.
3. *Declaring Order Dependencies.* In Section 3.3, we consider how OD information can be declared, and what types of natural ODs occur in today's schemas.
   (a) Order dependencies can be explicitly declared in our implementation in DB2 as a type of integrity constraint.
   (b) We demonstrate how ODs between surrogate and natural keys can be used for strong performance improvement [19].
4. *Inferring Order Dependencies.* In Section 3.4, we show how the optimizer can infer new ODs from known ODs. The known ODs may not match *interesting orders* in query planning, while ODs that logically derive from them would. Thus, such an OD-inference facility is ultimately needed to take

[3]The values for d_quarter are $1, \ldots, 4$ and for d_month, $1, \ldots, 12$.

Table 1: Notational conventions.

- **Relations**
  - **R** represents a *relation*, and **r** represents a specific *relation instance* (*table*).
  - A, B and C represent *attributes*.
  - s and t represent *tuples*.
  - t_A denotes the value of attribute A in tuple t.
- **Sets**
  - calligraphic letters denoted as $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ represent *sets* of attributes.
- **Lists**
  - bold letters represent *lists* of attributes: $\mathbf{X}$, $\mathbf{Y}$ and $\mathbf{Z}$. Note list $\mathbf{X}$ could be the empty list, $[\,]$.
  - square brackets denote an explicit list: $[A, B, C]$.
  - $[A \mid \mathbf{T}]$ denotes that A is the *head* of the list, and $\mathbf{T}$ is the *tail* of the list (the remaining list when the first element is removed).

fuller advantage of these techniques.

(a) We define a database to be *natural* if given order property over its attributes can be guaranteed. (All real-world domains we have encountered have this property, and thus are *natural*.)

(b) We discuss a general, efficient (polynomial) inference procedure which we have implemented which is sound and complete over *natural* domains.

In Section 4, we present results of a performance study over queries over TPC-DS.

5. *Experimental Results.*
   All nine of the test queries show a significant performance gain using the OD-extended optimizer, with an average 30% time improvement over a ten-GB database.

In Section 5, we discuss related work, both previous applied work that used dependencies in optimization (upon which we build), and theoretical work on order dependencies which has provided critical foundations for our current implementation. In Section 6, we outline next steps for this work, and conclude.

2. BACKGROUND

We adopt the notational conventions in Table 1. We are interested in *lexicographical ordering*, or *nested sort*, as is provided by SQL's order-by directive. Given order by $A_0, \ldots, A_{n-1}$ in a query, the answer tuples are first sorted by A_0; then, within any group (partition) of tuples with the same value for A_0, the tuples are sorted by A_1; within groups with the same A_0 value and A_1 value, by A_2; and so on.

DEFINITION 1. (operator '$\preceq_{\mathbf{X}}$') *Let $\mathbf{X}$ be a list of attributes, $\mathbf{X} = [A \mid \mathbf{Z}]$, and s and t be two tuples in relation instance $\mathbf{r}$. Define the binary* operator *("relation") '$\preceq_{\mathbf{X}}$' as $s \preceq_{\mathbf{X}} t$* iff *$s_A < t_A$* or *($s_A = t_A$ and ($\mathbf{Z} = [\,]$ or $s \preceq_{\mathbf{Z}} t$)).*

The default *direction* of the order for SQL's order-by is *ascending*. That is, order by $A_0, \ldots, A_{n-1}$ is equivalent to order by A_0 asc, $\ldots$, A_{n-1} asc. In this work, we do not consider order-by's with all *descending* (desc) directives, without loss of generality. We also do not consider order-by's that mix asc and desc directives; e.g., order by A asc, B desc.[4]

[4]In [17] and [21] we consider a mix of asc and desc orders. The inference problem for ODs with mixed directives is more computationally complex [21]. So this restriction is not "without loss of generality".

Table 2: An Instance of Table d_date.

d_date_sk	d_date	d_year	d_month	d_day	d_quarter
7300	20111230	2011	12	30	4
7301	20111231	2011	12	31	4
7305	20120105	2012	01	05	1
7306	20120106	2012	01	06	1
7333	20120201	2012	02	01	1

DEFINITION 2. (Order Dependency) *An* order dependency *(OD) is a dependency between two* lists *of attributes. We write an OD as $[A_0, \ldots, A_{m-1}] \mapsto [B_0, \ldots, B_{n-1}]$. A table—with attributes A_0, $\ldots$, A_{m-1} and B_0, $\ldots$, B_{n-1}—*satisfies *the OD iff any list of the table's tuples that satisfies order by A_0 asc, $\ldots$, A_{m-1} asc also satisfies order by B_0 asc, $\ldots$, B_{n-1} asc. That is, given table $\mathbf{r}$, $\mathbf{X} = [A_0, \ldots, A_{m-1}]$, $\mathbf{Y} = [B_0, \ldots, B_{n-1}]$, then*
$\forall s, t \in \mathbf{r}$. *$s \preceq_{\mathbf{X}} t$ implies $s \preceq_{\mathbf{Y}} t$.*

For $\mathbf{X} \mapsto \mathbf{Y}$, we say $\mathbf{X}$ orders *$\mathbf{Y}$. Define $\mathbf{X} \leftrightarrow \mathbf{Y}$* iff *$\mathbf{X} \mapsto \mathbf{Y}$ and $\mathbf{Y} \mapsto \mathbf{X}$. For $\mathbf{X} \leftrightarrow \mathbf{Y}$, we say $\mathbf{X}$ and $\mathbf{Y}$ are* order equivalent.

EXAMPLE 1. (ODs over an instance of the d_date table) *Order dependencies*
$[d\_date\_sk] \mapsto [d\_year, d\_month, d\_day]$ *and*
$[d\_year, d\_month, d\_day] \mapsto [d\_date]$
are satisfied in Table 2. On the other hand, order dependencies
$[d\_year, d\_month] \mapsto [d\_date]$ *and*
$[d\_date\_sk] \mapsto [d\_year, d\_day, d\_month]$
are falsified *by Table 2.*

Date and *time* are richly supported in the SQL standards. The widely-used benchmark standard TPC-DS contains 99 queries. Of these 99, 85 involve date operators and predicates and five involve time operators and predicates. If the idea of order dependency were only applicable to date and time, they could confer great benefit on query optimization. Our observations with IBM customers has borne this out. The vast majority of their queries over data warehouses do involve date and time attributes, and SQL functions and algebraic expressions over them. The concept behind ODs is not limited to the time domain, however. ODs occur in other domains arising from business semantics: sequence numbers, surrogate keys, and measured values such as sales, stock prices, taxes, human resources (We show examples of the last two domains examples in Examples 4 and 5, respectively.)[5] The TPC-DS benchmark is for *decision-support*, and its schema typifies that of data warehouses (DWs) designed to aid analysis of business over a historical period.

The schema is a common multi-dimensional model based on a star schema with fact and dimension tables. Fact tables will have many rows capturing measures or events over time, such as sales. Dimension tables model the entities such as the customers and products. Date is often made an explicit dimension table, because the designers need to keep specific data about given dates (e.g., a holiday indicator, a weekday indicator, the name of the holiday, and the name of special events). In the TPC-DS schema, the date dimension has a granularity of day. (See Figure 1.) Data kept about the entities is factored out into the dimension tables (and out of the fact table) based on good principles of design (*normalization*), but

[5]Of course, any OD implies a corresponding FD, modulo lists and sets, but not vice versa. Even so, the other direction (the vice versa) is often true; that attributes that are functionally related (as with FDs) are often order related (as with ODs).

also for practical reasons. Because the fact table will be very large row-wise, it is important to keep the size of rows small.

A common design question for DWs is whether to use *surrogate* keys [10]. SQL's *date* data type would be a good *natural* key in the date table. However, it used to be common in database systems—and still is in some—for the *date* data type to take eight bytes. Given a fact table with a billion rows, each byte per row is a gigabyte of storage. So instead, a four-byte integer could be used as a surrogate key in the date table; then, the fact table's foreign-key field referencing table date is smaller. This is the design choice in TPC-DS's schema.

Furthermore, the date dimension table can be populated at the time the DW is created. It does not undergo regular updates. Its surrogate key can be generated via increment, in the order of the date values. Therefore, there will be an order equivalence between the surrogate key and the date's day.

While the use of the surrogate key helps reduce the size of the fact table, it does introduce costs at query time. Queries will often have predicates involving (natural) date values to access data from the fact table. This necessitates a potentially expensive join between the fact and date tables.

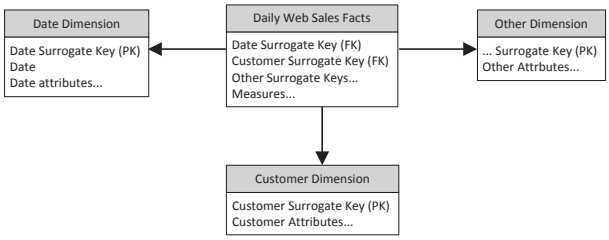

Figure 1: Standard TPC-DS schema.

IBM recommends to its business customers to use a natural key for the date table (using the *date* data type). IBM DB2 manages to store the *date* data type in a compact four bytes. The advantages of using a surrogate key in this case are nullified. For this reason, we consider also a variation of the TPC-DS schema as in Figure 2 which uses the natural date key in the date table and in the fact table for the foreign key.

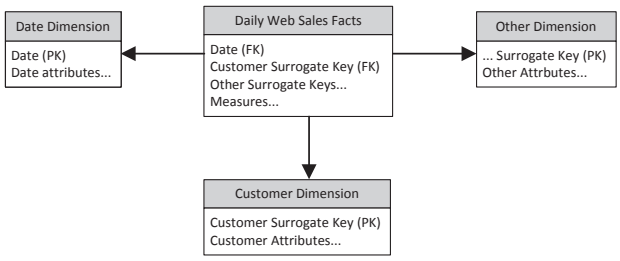

Figure 2: Alternative TPC-DS schema with natural date key.

For our performance study (Section 4), we test six queries under the unmodified TPC-DS schema (Figure 1), and three queries under the alternative schema (Figure 2). The queries are motivated and presented in the next section. The two schemas allow us to illustrate different optimizations using ODs that can be accomplished. It also shows that we achieve good performance improvements in both, so our techniques do not require specific schema designs.

3. OPTIMIZATION WITH ODS

In Section 3.1, we discuss how current query optimizers can be extended to use ODs. In Section 3.1.1, we delve more in depth into how *order* is presently used in optimization, and how ODs extend the effectiveness of this. In Section 3.1.2 we present two algorithms, *Reduce Order OD* and *Homogenize Order OD*, that extend broadly two existing algorithms in DB2 for matching interesting orders [16] further to accommodate ODs. Of course, for these techniques to add benefit, ODs must exist and be known. ODs can arise from three sources: they can be detected in the context of the query, they may be declared for the database as integrity constraints; and they may be inferred from known ODs.

Local order dependencies can arise within a query's scope, due to the query's semantics and constructs. For instance, if there is a predicate $\mathsf{A} = \mathsf{B}$ in the where clause, then clearly the OD $[\mathsf{A}] \leftrightarrow [\mathsf{B}]$ is satisfied in the query's scope (but is not necessarily satisfied generally in the database). Local ODs also arise through a query's *derived attributes* via SQL functions and algebraic expressions, as motivated in Section 1.1. The optimizer must *detect* local ODs to use them. In Section 3.2, we demonstrate the types of local ODs we have instrumented DB2 to detect and use, and we illustrate these with customer-motivated, real-world queries over the TPC-DS schema. These queries are used in our performance study, presented in Section 4. ODs can also be *declared* on a database as *integrity constraints*. In Section 3.3, we address this.

Even with the ODs declared for the database and the local ODs deduced within the scope of the query, the optimizer might miss opportunities. There may be an OD that logically follows from the declared and local ODs that would allow for a better plan, while none of the declared or local ODs match directly. For instance, again assume there is a predicate $\mathsf{A} = \mathsf{B}$ in the where clause. If we also knew the declared OD $[\mathsf{A}] \mapsto [\mathsf{Z}]$, within the query's scope, OD $[\mathsf{B}] \mapsto [\mathsf{Z}]$ is also satisfied (by *transitivity* of ODs [18]). Therefore, the optimizer has a need to *infer* ODs from others. In Section 3.4, we show when and where the optimizer would invoke OD-inference procedures (two of which were presented in Section 3.1.2 as algorithm, for canonicalization), and we develop such procedures.

To the best of our knowledge, we are the first to bring reasoning over order dependencies into the query optimizer of a relational database system.

3.1 Using ODs in the Optimizer

We motivate *order dependencies* in analogy to *functional dependencies*: FDs are to group-by as ODs are to order-by. Order is an additional property over a partition that plays important roles in databases.[6] ODs can be used to great advantage in query processing, just as FDs have been [16].

3.1.1 Order in Optimization

On the one hand, order is irrelevant in the relational model on the *logical* side. Relational instances are *sets* of attributes, and a schema is a *set* of attributes. So there is no notion of order. (For different data models such as XML, order is an integral part of the

[6]The group-by and order-by clauses in SQL are syntactically the same, excepting the optional sort-direction directives—ascending (`asc`) and descending (`desc`)—in order-by. Their meanings are quite similar too; order-by results in an ordered list (*ordering*) of the answer tuples, which matches the *partitioning* of the tuples with respect to an equivalent group-by. Of course, group-by's are over sets of attributes (as are FDs), while order-by's are over lists of attributes (as are ODs), as the order of attributes is important to specify the lexicographical order at the data. This difference makes working with ODs harder.

model itself.) SQL concedes a single order-by clause to be appended to a query to order the result set, as a convenience, given that people usually want to see the results organized in a given way. (The SQL extension of window aggregation provides this too.)

On the other hand, order plays an important role on the physical side, in storage, indexes, and optimization.

- *indexes.*
  Data is often referenced by (clustered) tree indexes, which provides ordered access.
- *pipelining.*
  In a query plan tree, *pipelining* is a prevalent technique. This is when a parent operator can *pull* its input streams from its child operators as they produce their (output) streams. The operator's procedure may need its input sorted in a given way, as does a *merge* join. An operator such as group-by or order-by can be handled very efficiently *on-the-fly* when its input stream is ordered appropriately. Pipelining between operators also saves processing and possibly I/O since the results of the child operator do not have to be fully *materialized*, *spilled* to disk with expensive I/O overhead.
- *interesting orders.*
  Some access paths and procedures will result in the operator's output stream being ordered. It may be that a procedure can be chosen for the parent operator which relies on this ordered stream for input and which is less expensive than the alternative choices.
  This enables pipelining between the operators, and may also be less expensive as it allows the optimizer to forgo inserting an expensive operation in between. For example, the operator in the tree under a group-by might provide its output ordered in such a way the group-by's partitioning can be done *on-the-fly*. If not, an expensive partitioning or sort operation has to be inserted into the tree. Interesting orders can be effective for *join*, *order-by*, *group-by*, *partition-by*, and *distinct*.

Say that we know the database satisfies $\mathbf{X} \mapsto \mathbf{Y}$. Given a query with order by $\mathbf{Y}$, we can rewrite it instead with order by $\mathbf{X}$. Note that, unless $\mathbf{X} \leftrightarrow \mathbf{Y}$, the original and rewritten query are not "semantically" equivalent! The rewritten query satisfies the intent of the original (but, perhaps, not vice versa). *Strengthening* the order-by conditions is allowed, but *weakening* them is not.

This is an important property for query plans with ordered tuple streams. It means order *equivalences* are not required for valid query rewrites; directional order dependencies (that is, $\mathbf{X} \mapsto \mathbf{Y}$ instead of $\mathbf{X} \leftrightarrow \mathbf{Y}$) suffice. This provides us with much versatility for rewrites.

Sorting is an expensive operator. The key goal of our OD-optimization is to optimize or eliminate sorting operations in query plans whenever possible. Our techniques are built upon the seminal techniques for order optimization from [16].

```
select year(a.y),...
from a, b
where a.x = b.x
group by year(a.y)
order by year(a.y);
```

Query 3: Template of the query.

Query 3 is a query sketch of a common pattern seen in analytic queries. It employs the SQL function year. Figure 3 illustrates a query plan one would expect for the query in Query 3. Let the index employed as the access path on table a be on its column y, a date type. A sort operator is placed under the group-by and order-by operators, regardless, as the optimizer does not recognize that a.y *orders* year(a.y). Our work is to recognize these order dependencies, to "remove" the sort operator from access plan.

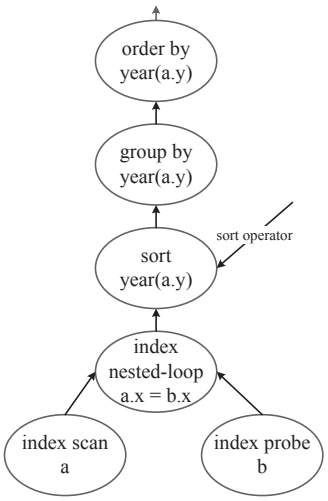

Figure 3: Query access plan.

3.1.2 Canonical Form

The critical role of *interesting orders* was recognized quite early [15]. Because we are interested in ordered streams between operators in the query plan (to allow for pipelining, selecting more efficient procedures, and eliminating intermediate sort and partitioning steps), the optimizer needs to track which stream orders are possible to generate by alternative sub-plans. The ones that the optimizer tracks during query plan construction are called *interesting orders*.

The optimizer needs to determine which orders that sub-plans can produce are "interesting"; an order is not interesting if it is of no potential benefit to any other operator.

In DB2, interesting orders are generated in a top-down scan of the graph of the query prior to the planning phase. (This is called the *order scan*.) Interesting orders are considered for join, order-by, group-by, partition-by and distinct operators. They are represented as lists of attributes. Interesting orders are *pushed down* and the DB2 optimizer tries to combine interesting orders whenever possible. As interesting orders are pushed down, they can turn into sort-ahead orders. (Sort-ahead allows a sort for something like order-by to be pushed into a join tree.) This process enables multiple interesting orders to be satisfied by one sort. Different alternative plans are tried and the least expensive is chosen.

Generating interesting orders is a complex task. Many different orders could apply for a given operation. For example, group by $\mathsf{A}_0, \ldots, \mathsf{A}_{n-1}$ can be accommodated on-the-fly by an input stream ordered by $[\mathsf{A}_{p_0}, \ldots, \mathsf{A}_{p_{n-1}}]$, for *any* one of the $n!$ permutations of $\{p_0, \ldots, p_{n-1}\} \subseteq \{0, \ldots, n-1\}$. Matching order specifications is also a complex task. For instance, group by A_1, A_2, A_3 can be done on-the-fly with an input stream ordered by $[\mathsf{A}_3, \mathsf{A}_1, \mathsf{A}_2, \mathsf{A}_4]$, but not by $[\mathsf{A}_3, \mathsf{A}_1, \mathsf{A}_4, \mathsf{A}_2]$.

On the one hand, the number of orders deemed "interesting" must be contained because of the sheer number of possibilities. On the other hand, we want to label more of those orders as "interesting" which would offer more planning options. In particular, we should recognize any order that is *order equivalent* with, or that *orders*, any order already marked as interesting.

One of the most basic operations used by order optimization is *reduction*. Reduction is the method of rewriting an order specification in a simple *canonical form*. This includes substituting each column in the specification by an equivalent one, and then removing all redundant columns. Reduction is fundamental for testing whether an order property satisfies an interesting order.

In [16], the authors explored the important role of *order* for optimizing queries. They introduced query rewrites in IBM DB2 that could exchange one interesting order by another, when it is known that the orders were *order equivalent* (as defined in this work). They employ functional dependencies for this very task. The group by A_1, A_2, A_3 can be done on-the-fly with the input A_3, A_1, A_4, A_2, if $\{A_1, A_2, A_3\} \rightarrow \{A_4\}$. In that case, orders $[A_3, A_1, A_4, A_2]$, and $[A_3, A_1, A_2]$ are order equivalent.

DEFINITION 3. (generated attribute) *A generated attribute is an attribute computed from other column using algebraic expressions and SQL functions.*

EXAMPLE 2. (attribute generated based on date and month) *Let G = year(d_date)*100 + month(d_month). Thus, G is a* generated attribute.

We extend further the techniques of [16] by also employing order dependencies to recognize more order optimization techniques. (Their rewrites rely on FD information available to the optimizer, but do not use *order dependencies*.)

Algorithm 1 Reduce Order OD

Input:
A set of ODs $\mathcal{M}$ and
order specification $\mathbf{O} = [O_0, O_1, ..., O_{n-1}]$.
Output:
The reduced version of $\mathbf{O}$.

```
1: for i ← 0 to n − 1 do
2:    if O_i is generated attribute from G and O_i ↔ G then
3:       O = [O_0, ..., O_{i−1}, G, O_{i+1}, ..., O_{n−1}]
4: Rewrite O in terms of each column's equivalence class head.
5: for i ← n − 1 to 0 do
6:    Let B = {O_0, ..., O_{i−1}}
7:    if order specification is a single (O = [O_0]), generated at-
      tribute from G and G ↦ O_0 then
8:       O = [G]
9:    else if B → O_i then
10:      Remove O_i from O
11:   else if [O_0, ..., O_{i−1}, O_{i+1}, ..., O_{n−1}] ↦ [O_0, ..., O_{n−1}]
      then
12:      Remove O_i from O
13: return O
```

They introduced an algorithm, *Reduce Order*, which traverses the interesting-order list of attributes from right to left, that checks to eliminate attributes. This is for putting interesting orders into a canonical form.[7] We extend this algorithm[8] by iterating through the list, additionally checking following.

- Whether the currently considered attribute O_i is a generated attribute from G and $O_i \leftrightarrow G$. If so, the attribute O_i is replaced by the attribute G in the list, $\mathbf{O} = [O_0, ..., O_{i-1}, G, O_{i+1}, ..., O_{n-1}]$.
- Whether the order specification is a single, generated attribute from an attribute G. (Call this attribute O_0.) If, $G \mapsto O_0$, the attribute O_0 is replaced by the attribute G in the list, $\mathbf{O} = [G]$. Detecting monotonicity property for generated attributes is described in detail in Section 3.2.
- Whether the *list* without the attribute being currently considered *orders* the full list. If so, the attribute is dropped from the current list.

With this, we can optimize queries such as Query 2 in Section 1. We infer

[d_year, d_month, d_day] ↦
[d_year, d_quarter, d_month, d_day]

from the set of declared ODs. Then, both the order-by and group-by can be reduced from d_year, d_month, d_quarter, d_day to just d_year, d_month, d_day. (Similarly, we can optimize Query 1 by detecting that d_date ↦ d_date + 30 days.)

EXAMPLE 3. (Canonical form with dependencies) *Consider statement group by d_date + 30 days, d_year, d_month. The canonical form is necessary to recognize that this is equivalent to group by d_date, d_year, d_month. It is only after that could reductions by FDs — d_date → d_year and d_date → d_month — be applied to reduce this even further to group by d_date.*

The algorithm *Reduce Order OD* is correct because removing O_i from the list using a FD $B \rightarrow O_i$ is part of *Reduce Order* algorithm described in [16] (Algorithm 1, lines 9–10). Given an order dependency $X \mapsto Y$, the clause order by Y, can be rewritten with order by X, as strengthening the order-by conditions is allowed as described in Section 2 (Algorithm 1, lines 7–8 and Algorithm 1, lines 11–12). It is also sound to replace order equivalent attributes ($O_i \leftrightarrow G$, Algorithm 1 lines 2–3).

Note that, when sorting is required, the simplified version of $\mathbf{O}$ provides a reduced number of sorting columns. This is important for minimizing sort costs. It may also happen that because of a reduced $\mathbf{O}$, an index can be matched, eliminating the need for a sort operator altogether.

In DB2, some columns might be substituted with *equivalent columns* in the new context. For instance, columns can be substituted with the ones on which an index is declared. This process is called *homogenization*. The *Homogenize Order* algorithm is described in [16]. It uses *equivalent classes* to substitute columns in an interesting order $\mathbf{O}$. We extend the algorithm as *Homogenize Order OD* to account for order dependencies.

The algorithm Homogenize Order OD is correct because, given an OD $A \leftrightarrow B$, if the data are ordered by A, they are also ordered by B (and vice versa), so A can be substituted by B in an interesting order $\mathbf{O}$ (Algorithm 2 lines 3–4).

3.2 Detecting Order Dependencies

Analytic queries often use functions, algebraic expressions, and case expressions. Order dependencies can be derived from built-in SQL functions, and from case expressions. For example, the SQL function *year* extracts the year component (the leading component) of the *date*. Thus, [date] ↦ [year(date)]. Let the table date_dim have an index on its d_date column. If it could detect the OD that d_date orders year(d_date), the optimizer could accomplish order by year(d_date) in a query by using an index scan over the d_date index to provide a correct "interesting" order, with no need to employ sort operation for it.

We describe our techniques using monotonicity properties. We evaluate them over queries designed based on experience with IBM

[7] The main body of their Reduce Order algorithm are lines 4 and 9−10 from Algorithm 1.

[8] In [18] we focused on order optimization techniques based on an axiomatization of ODs. In this work, we base the reduction on testing logical implication and algebraic expressions. This is more general and more efficient.

Algorithm 2 Homogenize Order OD

Input:
A set of ODs $\mathcal{M}$,
an interesting order O, and
a target order $\mathsf{T} = [\mathsf{T}_0, \mathsf{T}_1, ..., \mathsf{T}_{n-1}]$
Output:
O homogenized to T marked as O_T or
returned "false" indicating that O_T cannot be found.

1: Reduce O
2: Using $\mathcal{M}$ try to substitute each column in O from T
3: **if** for each A in O there exists B in T such that $\mathsf{A} \leftrightarrow \mathsf{B}$ **then**
4: **return** O_T
5: **else**
6: **return** "false"

customers. Our observations with customers queries at IBM have shown that queries that involve SQL functions and algebraic expressions involve unnecessary operators such as sort. Therefore, being able to optimize such queries could offer large benefits. These techniques have been implemented in DB2. The following rewrites were performed via this implementation. We describe how the monotonicity detection algorithm in IBM DB2 [11] allows for rewrites in the case of queries with order-by. We then show the value of this technique when combined with indexing in DB2. The algorithm detects monotonicity in algebraic expressions and SQL functions. It maintains a monotonicity state as the input expression is traversed. Given the parse tree of the expression to be checked, it answers whether the expression is monotonic (with respect to the attributes over which it is defined). It employs a transition table, scanning the left and right operands. For example, if the left side of the operand of sum operator is monotonic and right operator is a constant, the result is is also monotonic. (See [11] for details.)

In [11], they showed how to use this for predicate derivation which led to it being implemented in DB2. They offered no performance study, though. We demonstrate the value for SQL queries via interesting orders. In our implementation, the monotonicity detection algorithm is called during the query rewrite phase, when processing statements which involve *join*, *order-by*, *group-by*, *partition by*, and *distinct* (Reduce Order OD). This is useful for improving access methods (as discussed above), and also for improving cardinality estimation.

Monotonicity can be also detected for a variety of SQL built-in functions. Ones we demonstrate here include the following. Each is monotonic with respect to its input.

- year(): Returns the year of the date.
- substr(): Returns a sub-string of the string input. If the starting position of the requested substring in the string is one, the result is monotonic. For example substr(s_zip, 1, 2) is monotonic.
- concat(): Returns the concatenation of two strings. (When the first string is always the same length and the second string is a constant, the function is monotonic.)

Monotonicity is detected for a wide range of functions: functions that refer to time dimensions, such as day() and hour(); mathematical functions, such as log(), ceil(), and sqrt(); and type conversions, such as int() and float().

Query 4 employs the substring function in its group-by. Recall Query 1 in Section 1. We saw that a clever programmer could recompose it to avoid the performance problem that the use of the algebraic expression in the group-by and the order-by could cause. In this case, however, the programmer could not rewrite this to avoid the issue, since the substring changes the partition of the group-by.

Let there be an index on s_zip in table store. It is obvious that the column s_zip orders the derived column substr(s_zip, 1, 2). Given the optimizer detects this OD, it can choose to do an index scan using the index on s_zip to accomplish the group-by on-the-fly, and no partitioning or sort operator would be needed.

```
select substr(P.s_zip, 1, 2) as area,
       count(distinct P.s_zip) as cnt,
       sum(S.ss_net_profit) as net
from store_sales S, store P
where S.ss_store_sk = P.s_store_sk
group by substr(P.s_zip, 1, 2);
```

Query 4: Substring with group-by.

```
select I.i_item_desc,
       to_char(D.d_date,'YYYYMMDD')
           || ' 12:00:00' as when,
       sum(W.ws_sales_price) as total
from web_sales W, item I, date_dim D
where W.ws_item_sk = I.i_item_sk and
      I.i_category = 'Children' and
      W.ws_sold_date_sk = D.d_date_sk
group by I.i_item_desc,
         to_char(D.d_date,'YYYYMMDD')
             ||'12:00:00'
order by to_char(D.d_date,'YYYYMMDD')
             ||'12:00:00';
```

Query 5: With string conversion and concatenation.

Let there be an index on d_date in the date_dim table. In Query 5, the data are ordered by d_date converted to char and *concatenated* with a *time* constant, '12:00:00'. This is a type of query commonly used in business-intelligence reporting. The monotonicity detection algorithm works across the type conversion, and then over the string concatenation as the first string is known to be of a constant length. This makes the OD
[d_date] $\mapsto$ [to_char(d_date,'YYYYMMDD')||'12:00:00'],
visible to the optimizer.

Query 6 can be effectively rewritten by the optimizer to the form in Query 7. An evaluation of the rewritten query then uses the index on d_date. The two constants 1998 and 2002 are used in Query 6 as a filter predicate in its where clause. The optimizer could not use the index on d_date, however, since the predicate is over year(D.d_date). In the query rewrite, the filter is set on d_date, to be on the range between date('1998-01-01') and date('2002-12-31'). Then, the optimizer uses index on d_date in the query plan. The method above generalizes to a query rewrite technique. Note that this is not captured by canonical form, as formed by Reduce Order OD algorithm. This demonstrates a need for rewrite techniques with ODs beyond those that the canonicalization provides. It can be used for SQL functions (besides year() of course) or user defined functions. For example, let there be an index on an attribute A. Assuming, there is known OD [A] $\mapsto$ [sql_function(A)] (or [A] $\mapsto$ [user_defined_function(A)]), the predicate can be set accordingly over the attribute A instead of the calculated sql_function (or user_defined function) on A.

Query 8 is similar to Query 4, but with a filter predicate in its where clause. (This version does not contain a group-by clause, so the order-by could be rewritten manually. If a group-by were done on substr(H.w_warehouse_name,1,10) also, it could not be. It illustrates our OD techniques the same, though, in either case.)

Query 9 is an OLAP query that uses a *partition-by* clause over modified TPC-DS schema as in Figure 2. The query plan can em-

```
select I.i_item_desc, I.i_category,
       I.i_class, I.i_current_price,
       sum(W.ws_ext_sales_price) as revenue
from web_sales W, item I, date_dim D
where W.ws_item_sk = I.i_item_sk and
      W.ws_sold_date_sk = D.d_date_sk and
      year(D.d_date) between 1998 and 2002
group by I.i_item_id, I.i_item_desc,
         I.i_category, I.i_class,
         I.i_current_price
order by I.i_category, I.i_class,
         I.i_item_id;
```

Query 6: With the predicate *year*.

```
select ... from ... where ... and
       d_date between
           date('1998-01-01') and
           date('2002-12-31')
group by ... order by ...;
```

Query 7: Rewrite of Query 6.

ploy the index on ws_sold_date, if the optimizer detects via the monotonicity detection algorithm that OD

$$[\mathsf{ws\_sold\_date}] \mapsto [\mathsf{year(ws\_sold\_date)*100} + \mathsf{month(ws\_sold\_date)}]$$

holds, which effectively partitions by year concatenated with month.

Our work on order dependencies can be combined with notion of *near–sortedness*. If a stream is sorted by A, it may be *nearly sorted* on, say, year(A), B, C. If there is an index on A and it is known that every partition of year(A) is small, the stream could be produced by an index scan, and thus be ordered by A. Given $[A] \mapsto [\mathsf{year(A)}]$ and that the year(A)– blocks are small, each year(A)–block can be re–sorted in main memoery on–the–fly. This removes the need for an external sort operator. This technique also extends beyond the canonicalization.[9]

As an example, consider Query 10 with a case expression. The monotonicity detection algorithm is triggered due to the order-by statement. It detects that

$$[\mathsf{d\_date}] \mapsto [\mathsf{year(d\_date)}].$$

Therefore, the optimizer can then take advantage of the index on d_date, speeding up the sort operator in the plan, to accomplish the order-by.

3.3 Declaring Order Dependencies

In [19], we demonstrated that dramatic gains in query performance can be had in queries by recognizing ordering correspondences between attributes. Our techniques looked promising to generalize to many more types of the queries, which lead to the work here.

Most queries in a data warehouse are over the fact table. As discussed in Section 2, and as in TPC-DS's schema, surrogate keys are used in the dimension tables, and so for the foreign key columns in the fact table. A query often uses natural date values in its predicates, however. This requires a potentially expensive join between the fact and the date dimension tables.

When the fact table has been partitioned by date over many nodes

[9]Similarly, ODs and near-sortedness can be used when using SQL functions such as concat(), when the first string is of fixed length and the second string is not constant. Without the second string a constant as in Query 5, we can still use the index on the first string to provide a *prefix order* and then use "mini-sorts" on–the–fly to avoid spilling to the disk.

```
select substr(H.w_warehouse_name,1,10)
from web_sales W, warehouse H
where W.ws_warehouse_sk = H.w_warehouse_sk
      and W.ws_quantity > 90
order by substr(H.w_warehouse_name,1,10);
```

Query 8: Substring variation with order-by.

```
select count(*) as count
       over (partition by
          year(S.ws_sold_date_sk)*100
          + month(S.ws_sold_date_sk))
from web_sales S;
```

Query 9: OLAP query.

(as the fact table can be very large), this can be especially expensive. Since the date range (surrogate values) over the fact table cannot be determined from the query (natural values), all partitions of the fact table must be scanned. We optimize such queries involving dates by removing the join, and choosing just the relevant partitions of the fact table.

Query 11 from the TPC-DS benchmark, requires that expensive join between the fact table web_sales and the dimension table date_dim. The surrogate (date) keys in the date dimension table are ordered in the same way as natural date values in the dimension table, however. So there is a known order dependency between them. This can be declared as a check constraint in DB2.[10] This can be done whenever the database administrator knows of relevant order dependencies that are essentially a part of the *semantics* of the database. Declaring an OD as an integrity constraint gives a guarantee that the database will satisfy it. Thus, as $\mathsf{d\_date\_sk} \mapsto \mathsf{d\_date}$, two probes can be made into the dimension table to calculate the range of the surrogate keys in the fact table, finding the mindate and maxdate surrogate keys. These minimum and maximum surrogate values then replace the predicate in the where clause with the natural date values, so no join with the date dimension table is needed. Query 11 can then be simplified into the form shown in Query 12. (Details of when and how this rewrite can be performed in a general case appear in [19].)

We performed experiments over TPC-DS in our implementation in DB2 to demonstrate the efficiency of the approach. Thirteen of TPC-DS's queries matched for the rewrite. Each benefited, with an average performance gain of 48%.

3.4 Inferring Order Dependencies

In the sections above, we have discussed from where order dependencies arise. They can be declared explicitly as integrity constraints on the database. We have shown how the system can use these. Within the scope of a query, *local* ODs can arise from logical constraints in the query. We have shown how these can be detected and used. Lastly, ODs can logically follow from known ODs. For our techniques to be most effective then, the optimizer needs an inference capability for ODs.

We present an efficient inference procedure for ODs which is *sound* and *complete* over *natural domains*. (We define formally this *natural* property of domains in Definition 6 below. All the domains we see in practice and that we have used in this paper, such as the TPC-DS schema, are *natural*.) We have implemented this solver in IBM DB2 V10.

As discussed earlier, order dependencies are not limited to date

[10]In our implementation in DB2 in [19], we had a way to express ODs internally, whereas in this work we propose to declare such dependencies as formal integrity constraints.

```
select year(D.d_date), M.sm_type, S.web_name,
       sum(case when
                (W.ws_ship_date_sk
                    - W.ws_sold_date_sk <= 30)
            then 1 else 0 end) as "30 days",
                :
       sum(case when
                (W.ws_ship_date_sk
                    - W.ws_sold_date_sk > 120)
            then 1 else 0 end) as ">120 days"
from web_sales W, warehouse H, ship_mode M,
     web_site S, date_dim D
where W.ws_ship_date_sk = D.d_date_sk and
... group by
year(D.d_date), M.sm_type,
     S.web_name
order by year(D.d_date), M.sm_type,
     S.web_name;
```

Query 10: year(d_date) variation with order-by.

```
select ...
from web_sales W, item I, date_dim D
where W.ws_item_sk = I.i_item_sk and
      I.i_category
         in ('Sports', 'Books', 'Home') and
      W.ws_sold_date_sk = D.d_date_sk and
      D.d_date between
          cast('1999-02-22' as date) and
          (cast('1999-02-22' as date)
               + 30 days)...;
```

Query 11: With an expensive join.

Table 3: Table taxes.

id	salary	percent	taxes	group	subgroup
100	5000	19%	950	A	II
101	6000	19%	1140	A	III
102	3000	19%	570	A	I
103	20000	30%	6000	B	I
104	50000	40%	20000	C	I

and time. They commonly arise in many other domains.

EXAMPLE 4. (Taxes) *Consider table* taxes *in Table 3, which has columns for the taxable* salary*, tax* group*, tax* subgroup*,* taxes *on the salary, and the tax's* percent *of the salary. The tax groups are based on the level of salary and, therefore, increase with the salary. (The tax subgroup increases for the same group as the salary goes up, but oscillates within a group.) Assume that the taxes go up with income and are calculated by as a percentage. Thus, we can declare*

$[salary] \mapsto [taxes]$,
$[salary] \mapsto [percent]$, *and*
$[salary] \mapsto [group, subgroup]$.

It logically follows from these ODs that

$[salary] \mapsto [taxes, percent, group, subgroup]$.

This OD was derived automatically using our inference procedure for ODs described below.

Let the table taxes in Table 3 have a clustered index on salary. A query with order by taxes, percentage, group, subgroup given the three ODs as declared in Example 4 could then be evaluated using the index on salary, as the inference procedure could infer that

$[salary] \mapsto [taxes, percent, group, subgroup]$.

```
select ... from web_sales W, item I,
     (select min(d_date_sk) as mindate
         from date_dim
         where d_date >=
                 cast('1999-02-22' as date))
         as A,
     (select max(d_date_sk) as maxdate
         from date_dim
         where d_date <=
                 cast('1999-02-22' as date)
                   + 30 days)
         as Z
where ... and
      W.ws_sold_date_sk between
          A.mindate and Z.maxdate...;
```

Query 12: Rewrite of Query 11.

Obviously, the database administrator could have declared that OD too; but that is unlikely.

In Section 3.3, we had assumed that $[date\_sk] \mapsto [date]$. was declared. Instead, however, we may have had the following ODs:

$[date\_sk] \mapsto [year, month, day]$, and
$[year, month, day] \mapsto [d\_date]$.

From these, $[date\_sk] \mapsto [date]$ can be concluded.

The optimizer needs the means to discover ODs that logically follow from known ODs to benefit most from our techniques. We present an inference procedure as a formal means to do this.

Our inference procedure for testing logical implication of ODs is efficient, and is *sound* and *complete* over *natural domains*. The details of the proofs can be found in [21].[11]

We call an attribute a *constant* if, in any table that satisfies the set of ODs, it can have only a single value occurring in the table.

DEFINITION 4. (constant) *An attribute A is called a* constant *with respect to $\mathcal{M}$* iff $\mathcal{M} \models [] \mapsto A$.

DEFINITION 5. (order compatible) *Two lists $\mathbf{X}$ and $\mathbf{Y}$ are* order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$,* iff $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

In [21], we have shown that order dependency $\mathbf{X} \mapsto \mathbf{Y}$ can be falsified in two ways: $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{X} \sim \mathbf{Y}$.

We observe that a relation satisfying the OD $\mathbf{X} \leftrightarrow \mathbf{Y}$ satisfies the OD $\mathbf{X} \sim \mathbf{Y}$, but conversely a relation which satisfies the OD $\mathbf{X} \sim \mathbf{Y}$ may not necessarily satisfy the OD $\mathbf{X} \leftrightarrow \mathbf{Y}$. The following example helps to illustrate that point.

EXAMPLE 5. (Difference between two lists being order compatible and order equivalent.)
Order dependency

$d\_month \sim quarter$

is satisfied by table date_dim*. On the other hand, order dependency*

$d\_month \leftrightarrow quarter$

is falsified by table date_dim*.*

It is surprising perhaps, that the *order-compatibility* relation ('$\sim$') is *not* transitive. It is simple to show that the *orders* relation ('$\mapsto$') is transitive.

EXAMPLE 6. (Order compatibility is not transitive.) *Assume $\mathcal{M} = \{A \sim B, B \sim C\}$. The table in Table 4 satisfies the set of ODs $\mathcal{M}$. It falsifies $A \sim C$, however. This demonstrates that the order-compatibility relation is* not *transitive.*

[11]The OD-inference problem over general domains is harder. In [21], we prove that the testing logical implication for ODs in general is co-NP-complete.

Table 4: Showing Lack of Transitivity.

A	B	C
0	0	1
1	0	0

If we restrict our domain to have a property that guarantees a limited form of transitivity over *order-compatibility*, then we can make an efficient inference procedure for ODs.[12] The property we prescribe is *transitivity of order compatibility over single attributes*. We call a domain *natural* if it satisfies this property.

DEFINITION 6. (natural domain) *A domain (relation schema) is* natural *iff it can be guaranteed that, for each relation* $\boldsymbol{R}$ *in the schema, for any three attributes A, B, and C, where B is not a constant, if* $A \sim B$ *and* $B \sim C$*, then* $A \sim C$*. If a domain has this property, we call it a natural domain.*

Assume there is an additional column d_trimester[13] in the table date_dim. The values of the attribute d_trimester divides year into three four-month periods, while those of d_quarter divides it into four three-month periods.

EXAMPLE 7. (Transitivity over order compatibility.)
Order compatibilities

quarter $\sim$ month *and*
month $\sim$ trimester

are satisfied in Table 2 by date domain. Also, so is

quarter $\sim$ trimester

Hence, the transitivity property holds over order compatibility.

All of the real-world business domains we have explored including the IBM customers schemas, TPC-DS schema, and the examples which are used in this paper (the Date, Tax and Salary domain), are *natural*, by Definition 6. To break the underlying property in data seems to be only by contrivance. For this reason, we have named this restricted domain *natural*, as it covers the cases one sees in practice. A database can be tested for naturalness in a straightforward way, by enumeration.

Let $\mathcal{M} = \{m_0, ..., m_{n-1}\}$ be a set of ODs defined over the set of attributes $\mathcal{U} = \{\mathsf{A}_0, ..., \mathsf{A}_{m-1}\}$. The set $\mathcal{M}$ is represented as a string of pairs, each pair representing an OD (the left-hand and right-hand sides of the dependency). Each side is a list of attributes. Let the length of the representation of $\mathcal{M}$, the string of concatenated left-hand and right-hand sides, be denoted by $|\mathcal{M}|$.

Let *m* be the OD $\mathbf{X} \mapsto \mathbf{Y}$, for which both $\mathbf{X}$ and $\mathbf{Y}$ are defined over the set of attributes $\mathcal{U}$. We denote the length of $\mathbf{X}$ and $\mathbf{Y}$ as $|\mathbf{X}|$ and $|\mathbf{Y}|$, respectively.

DEFINITION 7. ($\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$) *The problem of* testing logical implication *for ODs is, given a set of ODs* $\mathcal{M}$ *and a OD* $\boldsymbol{X} \mapsto \boldsymbol{Y}$*, to decide whether* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$*.*

We show a *sound* and *complete* (Theorem 1) algorithm for testing logical implication of ODs over *natural domains* which runs in polynomial time (Theorem 2) [21]. Our proof is based on the property that $\mathbf{X} \mapsto \mathbf{Y}$ *iff* $\mathbf{X} \mapsto \mathbf{XY}$ and $\mathbf{X} \sim \mathbf{Y}$. The first part $\mathbf{X} \mapsto \mathbf{XY}$ is true *iff* the functional dependency $\mathcal{X} \rightarrow \mathcal{Y}$ holds, for any list $\mathbf{X}$ that order the attributes from set $\mathcal{X}$ and list $\mathbf{Y}$ and set $\mathcal{Y}$, respectively.

[12]It is this lack of transitivity over the order-compatible relation generally that is at the heart of the complexity for the inference problem over general domains [21].

[13]The values for d_trimester are $1, 2, 3$.

Table 5: Table Employee_salary.

id	position	rank	grade	hire_date	salary	years
100	Manager	70	87%	20010112	80K	11
150	Secretary	30	90%	20050112	40K	7
200	Manager	70	90%	20060817	50K	6
202	Director	90	50%	20080817	200K	4
203	Director	90	95%	20080818	200K	4

We have shown that this can be verified in linear time [21]. We show how to test whether $\mathcal{M}$ implies $\mathbf{X} \sim \mathbf{Y}$. (The proof is based on the property of transitivity of order compatibility over single attributes, as defines natural domains in Definition 6.) These parts combine to complete the proof of soundness and completeness of our inference procedure for ODs over natural domains.

THEOREM 1. *(soundness and completeness) [21] Algorithm for testing logical implication* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$ *for ODs is sound and complete over natural domains.*

THEOREM 2. *[21] (complexity) Testing logical implication for transitive domains of ODs, (that is, whether* $\mathcal{M} \models \boldsymbol{X} \mapsto \boldsymbol{Y}$*) is solvable in polynomial time,* $O(|\boldsymbol{X}||\boldsymbol{Y}||\mathcal{M}|)$.

In [12], an axiomatization for order dependencies (defined as we have in this paper) over a restricted domain is presented. The authors called these *temporal functional dependencies* (TFDs), focusing on the time domain, and they provided an axiomatization for TFDs. A TFD $\mathbf{X} \rightarrow \mathcal{Y}$ means that $\forall\, \mathsf{A} \in \mathcal{Y}.\ \mathbf{X} \mapsto [\mathsf{A}]$, in which $\mathbf{X}$ describes time and the attributes in $\mathcal{Y}$ are time variants.

The domain is too restricted, unfortunately, to be of use for us. It effectively restricts one to ODs of the form that has just a single attribute in the right-hand side (e.g., $\mathbf{X} \mapsto [\mathsf{A}]$). In many of our examples, as in Examples 1 and 4, we need ODs with lists of multiple attributes on the right-hand side. Thus, TFDs do not suffice. Furthermore, no inference procedure for TFDs was defined.

We illustrate the use of ODs, and how they can be used to support SQL functions and user-defined functions. Consider the human-resource table employee_salary in Table 5.

EXAMPLE 8. (Human Resource) *A table employee_salary has the following attributes: id(employee identifier), position (corporate title), rank (ranking of the employee)*[14]*, grade (employee evaluation), hire_date (date hired), salary (employee's salary), years (years of service).*

Salary rises as ranking, years of service, and grade rise, in that lexicographical order. That is, $[rank, years, grade] \mapsto [salary]$*. Moreover, the date when the person was hired is monotonic with respect to the identifier:* $[id] \mapsto [date\_hire]$*. Assume these order dependencies are declared as check constraints. The first constraint which expresses that* $[rank, years, grade] \mapsto [salary]$ *may be used to check the consistency of the database. (It would detect errors in assigning the salary, according to the business logic.) Furthermore, assume the table has a clustered index on hire_date. Given a business query with order by date_hire, it could be evaluated using the index on id. Note also that an OD* $[id] \mapsto [date\_hire]$*can be used to save disk space, since no index on date_hire is needed.*

In this example, all of the inferences can be automatically done by our OD-inference procedure.

[14]The higher the position, the higher the rank. (For example, for the secretary, the rank is 30, while for the director, it is 90.)

4. EXPERIMENTS

We have implemented and tested in DB2 V10 the query rewrites described in Section 3. This section reports the performance of the six queries described above. Three more queries were run against a corresponding alternative database, based on the alternative schema (with the natural date key in the date table) as shown in Figure 2. This included variations of Queries 1 and 5 (Q1′ and Q5′, respectively), modified to match the alternative schema, and Query 9.

The experiments were performed on a performance testing machine with the operating system AIX 6.1 TL6 SP5 with four processors (Intel(R) Xeon(R) CPU) and 1GB of memory. Results were obtained on a ten-GB TPC-DS database.

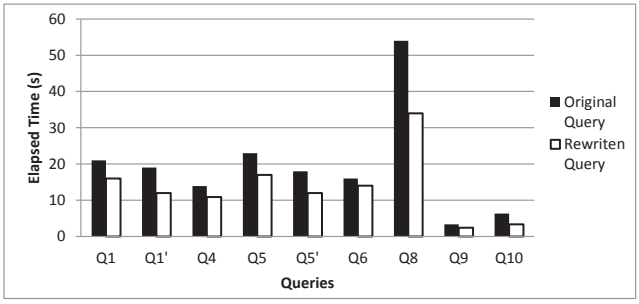

Figure 4: Performance results.

Figure 4 shows the execution times for the nine queries modified from the TPC-DS benchmark and expressing real world IBM customer scenarios, executed in two modes, with and without the OD-rewrites in the optimizer. Each query was run three times in both modes. (We repeat the tries in order to eliminate noise, including cold runs.) As shown in Figure 4, the results for the OD-optimized queries are significantly better. The performance improvement is, on average, a 30% improvement on elapsed time. Each of the nine queries benefited from the OD-rewrites. The improved efficiency is dependent on the precise nature of the database tables, the generated columns and the query itself. Our techniques eliminate and optimize expensive operations such as sort, which are super-linear, and which begin to dominate the execution costs as the database size increases.

5. RELATED WORK

Sorting is at the heart of many database operations: sort-merge join, index generation, duplicate elimination, ordering the output through the SQL order-by operator, etc. The importance of sorted sets for query optimization and processing was recognized very early on. Right from the start, the query optimizer of System R [15] paid particular attention to *interesting orders* by keeping track of all such ordered sets throughout the process of query optimization. In [9] authors explored the use of sorted sets for executing nested queries. In System R, interesting orders were primarily employed to prevent subplans that satisfy some useful order from being pruned by less expensive but unordered subplans during bottom-up plan generation. A later paper on the System R optimizer [1] shows how to combine interesting orders when possible from order-by, group-by, and distinct statements, so a single sort can be used more often to satisfy more than one operator. An other, strongly related paper is [15]. Its main contribution was a set of fundamental operations for use in order optimization by exploiting functional dependencies.

The importance of sorted sets has prompted the researchers to look *beyond* the sets that have been explicitly generated. Thus, [11] shows how to discover sorted sets created as generated attributes via algebraic expressions to use them for predicate derivation. In [19], we show how ODs discovered by reasoning over the physical schema can be used in query optimization.

Ordered sets and lattices have been a subject of research in mathematics [5]. Our concept of ODs is equivalent to *order-preserving mappings* between two ordered sets. The work in mathematics has concentrated on investigating properties of, and relationships between, ordered sets rather than among the mappings. To the best of our knowledge, no inference system for describing relationship between mappings of ODs has been proposed, besides ours.

Order dependencies were introduced for the first time in the context of database systems in [7]. However, the type of orders, hence the dependencies defined over them, were different from the ones we presented here. A dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ holds if the order over the values of *each* attribute of $\mathcal{X}$ implies an order over the values of *each* attribute of $\mathcal{Y}$.[15] In other words, the dependency is defined over the sets of attributes rather than lists.) The distinction between these two types of dependencies was later [12] aptly described as *point-wise* versus *lexicographical* order dependencies. Formally, an instance of a database satisfies a point-wise order dependency $\mathcal{X} \rightsquigarrow \mathcal{Y}$ if, for all tuples s and t, for every attribute A in $\mathcal{X}$, s[A] *op* t[A] implies that for every attribute B in $\mathcal{Y}$ s[B] *op* t[B], where *op* $\epsilon\{<, >, \leq, \geq, =\}$. In [7], a sound and complete set of inference rules for such dependencies is defined together with an analysis of the complexity of determining logical implication. A practical application of the dependencies for an improved index design is presented in [6].

Dependencies defined over lexicographically ordered domains were introduced in [12] under the name *lexicographically ordered functional dependencies*. Two other papers, [13] and [14], by the same author develop a theory behind both lexicographical as well as point-wise dependencies. (The latter were a simpler class than that defined in [7].) A set of inference rules (proved to be sound and complete) is introduced for point-wise dependencies, but not for lexicographical dependencies. Only a chase procedure is defined for the latter. In [18, 20], we presented a sound and complete axiomatization for ODs. We studied the expressiveness and (inference problem) complexity of ODs in [21].

A novel integrity constraint for ordered data, sequential dependencies (SDs), which are defined also over *sets*, of attributes was introduced in [8]. For example, an SD $\mathsf{sequence\_id} \rightsquigarrow_{[5,6]} \mathsf{time}$ means that time *gaps* between consecutive sequence numbers are between 5 and 6. The authors present a framework for discovering which subsets of the data obey prescribed sequential dependencies. An interesting study of establishing whether a given stream is sufficiently nearly-sorted was described in [2]. (An approach in [23] also works when order correlations are not perfect.)

6. CONCLUSIONS AND FUTURE WORK

Ordering permeates databases, to such an extent that we take it for granted. It appears in many queries and is relatively expensive to perform. Queries that involve order by, group by, join, partition by and distinct statement with SQL functions and algebraic expressions are common in real business scenarios. Identifying an order dependency between the attributes of such queries can remove or simplify potentially expensive operators such as sort. One of the paper's key contributions is an algorithm for reducing an *interesting order* into a *canonical form* by using *declared*, *detected*, and *inferred* ODs.

[15]For simplicity, we use the arrow $\rightsquigarrow$ for different type of orders.

The techniques described in this paper, although implemented in IBM DB2 V10, are general enough to be used in any query optimizer. These techniques should apply to a wide range of BI queries. Our experiments show the viability of our proposed solutions.

In future work, we plan to pursue following.

- Integrity constraints have been shown to be useful in query optimization via *query rewrites*. Functional dependencies are used to simplify queries with group-by operations [16], while inclusion dependencies are employed to remove redundant joins over primary and foreign keys [3]. We would further investigate how order dependencies can be useful in query optimization. This includes monotonicity in case expressions and optimization of queries such as Query 13 where there is an order dependency between the customer_id and the output of the then statement.

```
select ..., sum(S.quantity),
     (case
      when S.customer_id between 1 and 10
          then 1
               ⋮
      when S.customer_id between 91 and 100
          then 10
      end)
from sales S, ... where ...
group by (case ...) order by(case ...);
```

Query 13: Categories by case.

- We believe that ODs can also be identified in *geo-spatial* dimensions of DWs. Similar optimization techniques which we have described in this paper can be applied there, too.
- We want to improve the integration of order constraints with the cost-based optimizer to improve cardinality estimation. For example, when we know there is an order equivalence between columns, such as between d_date_sk and d_date, a surrogate and natural key, *and* we know there is a one-to-one mapping between them, then the cardinality of a range predicate on one could be estimated using the other. This could improve the performance beyond what we have already gained with our query-rewrite techniques.
- We are working on introducing a framework for discovering *conditional order dependencies*. (Conditional sequential dependencies were proposed in [8].) A conditional order dependency can be represented as a pair $(\mathbf{X} \mapsto \mathbf{Y}, \mathbf{T}_r)$, where $\mathbf{X} \mapsto \mathbf{Y}$, referred to as the embedded OD, and $\mathbf{T}_r$ is a range pattern tableau defining over which rows the dependency applies. A new type of integrity constraint could allow one to express that an OD holds within a sub-table.
- We plan to devise algorithms for data cleaning for ODs, similarly as in [4] and [22] for functional dependencies.

Acknowledgments

IBM and DB2 are registered trademarks of International Business Machines Corporation in the United States, other countries, or both. A current list of IBM trademarks is available on the Web as "Copyright and Trademark Information" at http://www.ibm.com/legal/copytrade.shtml.

7. REFERENCES

[1] G. Antoshenkov. Query processing in DEC rdb: Major issues and future challanges. In *IEEE Bulletin on the Technical Comittee on Data Engineering, 42-52*, 1993.

[2] S. Ben-Moshe, Y. Kanza, E. Fischer, A. Matsliah, M. Fischer, and C. Staelin. Detecting and exploiting near-sortedness for efficient relational query evaluation. In *ICDT, 256-267*, 2011.

[3] Q. Cheng, J. Gryz, F. Koo, T. Leung, L. Liu, X. Qian, and K. Schiefer. Implementation of Two Semantic Query Optimization Techniques in DB2 Universal Database. In *VLDB, 687-698*, 1999.

[4] F. Chiang and R. J. Miller. A Unified Model for Data and Constraint Repair. In *ICDE, 446-457*, 2011.

[5] B. Davey and H. Priestley. Introduction to Lattices and Order. In *Cambridge University Press, 1-298*, 2002.

[6] J. Dong and R. Hull. Applying Approximate order dependency to Reduce Indexing Space. In *SIGMOD, 119-127*, 1982.

[7] S. Ginsburg and R. Hull. Order dependency in the Relational Model. *Theoretical Computer Science, 149-195*, 1983.

[8] L. Golab, H. Karloff, F. Korn, A. Saha, and D. Srivastava. Sequential Dependencies. *PVLDB, 2(1): 574-585*, 2009.

[9] R. Guravannavar, H. Ramanujam, and S. Sudarshan. Optimizing Nested Queries with Parameter Sort Orders. In *VLDB, 481-492*, 2005.

[10] R. Kimball and M. Ross. The Data Warehouse Toolkit Second Edition. The Complete Guide to Dimensional modeling. In *John Wiley and Sun*, 2012.

[11] M. Malkemus, P. S., B. Bhattacharjee, L. Cranston, T. Lai, and F. Koo. Predicate Derivation and Monotonicity Detection in DB2 UDB. In *ICDE, 939-947*, 2005.

[12] W. Ng. Lexicographically Ordered Functional dependencies and Their Application to Temporal Relations. In *IDEAS, 279-287*, 1999.

[13] W. Ng. Ordered Functional Dependencies in Relational Databases. *Informaiton Systems, 535-554*, 1999.

[14] W. Ng. An Extension of the Relational data model to incorporate ordered domains. *ACM Transactions Database Systems, 344-383*, 2001.

[15] P. Selinger and M. Astrahan. Access Path Selection in a Relational Database Management System. In *SIGMOD, 23-34*, 1979.

[16] D. Simmen, E. Shekita, and T. Malkemus. Fundamental Techniques for Order Optimization. In *SIGMOD, 57-67*, 1996.

[17] J. Szlichta, P. Godfrey, and J. Gryz. Chasing Polarized Order Dependencies. In *AMW, 168-179*, 2012.

[18] J. Szlichta, P. Godfrey, and J. Gryz. Fundamentals of Order Dependencies. *PVLDB 5(11): 1220-1231*, 2012.

[19] J. Szlichta, P. Godfrey, J. Gryz, W. Ma, P. Pawluk, and C. Zuzarte. Queries on dates: fast yet not blind. In *EDBT 497-502*, 2011.

[20] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Axiomatic System for Order Dependencies. In *AMW*, 2013.

[21] J. Szlichta, P. Godfrey, J. Gryz, and C. Zuzarte. Expressiveness and Complexity of Order Dependencies. *PVLDB 6(14): 1858-1869*, 2013.

[22] M. Volkovs, F. Chiang, J. Szlichta, and R. J. Miller. Continuous Data Cleaning. In *ICDE (12 pages)*, accepted to appear in 2014.

[23] X. Wang and M. Cherniack. Avoiding Sorting and Grouping In Processing Queries. In *VLDB, 826-837*, 2003.