Business-Intelligence Queries in DB2 with Order Dependencies

Jaroslaw Szlichta, Parke Godfrey, Jarek Gryz, Wenbin Ma, Weinan Qiu and Calisto Zuzarte

Technical Report CSE-2012-04

July 24 2012

Department of Computer Science and Engineering
4700 Keele Street, Toronto, Ontario M3J 1P3 Canada

Business-Intelligence Queries in DB2 with Order Dependencies

Jaroslaw Szlichta[*,†,1] Parke Godfrey[*,†,2] Jarek Gryz[*,†,3] Wenbin Ma[‡,4] Weinan Qiu[‡,5] Calisto Zuzarte[†,‡,6]

[*]*York University*
Toronto, Canada
[†]*Centre for Advanced Studies*
IBM Toronto
Toronto, Canada

[1]jszlicht@cse.yorku.ca
[2]godfrey@cse.yorku.ca
[3]jarek@cse.yorku.ca

[‡]*IBM Toronto Laboratory*
Toronto, Canada
[4]wenbinm@ca.ibm.com
[5]davidqiu@ca.ibm.com
[6]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 has come to be called an *order dependency* (OD). Queries can be ran faster when the query optimizer uses order dependencies. They can be ran 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 inferred 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* and eliminate or simplify potentially expensive sorts in the query plan. We evaluate these techniques over our implementation in IBM® DB2® V10 using TPC-DS® benchmark. 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 is sound and complete over sets of ODs over *natural domains*. This enables the optimizer to infer useful ODs from known ODs.

I. Introduction

A. Motivation

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

These subtleties cause the optimizer to miss opportunities to use indexes 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 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. (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.) 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). It is akin to the well-known concept of *functional dependencies* (FDs) [1]. (In fact, any OD is inherently also an FD, but not vice versa.)

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

Consider then the SQL query in Query 1 over the TPC-

```
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.

```
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_id = D.d_date_id 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.

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 table has columns 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.

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 was 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.

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 [3], 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.

This FD, d_month $\rightarrow$ 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. Because d_month *orders* d_quarter—which says more than just that d_month *functionally determines* d_quarter—d_quarter can be removed from the order-by clause also, to result in a semantically equivalent query.[1] In this work, we show how this is accomplished.

B. Contributions and Outline

In Section II, 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 III, 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.*
   In Section III-A, we go into further depth how ODs are used to optimize.
2) *Declaring Order Dependencies.*
   In Section III-B, we consider how OD information can be declared, and what types of natural ODs occur in today's schemas.
   a) Order dependencies can be 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. (This was some of our preliminary work in this area, and a full version of this appears in [4].)
3) *Detecting Order Dependencies.*
   In Section III-C, we show how ODs between columns and functions over columns (user-defined functions and algebraic expressions) can be automatically detected by the optimizer.
   a) These techniques have been implemented within DB2.

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

b) We present a suite of real-world IBM customer queries over TPC-DS that illustrate the issues, which are then used in Section IV for an experimental performance evaluation. The optimizer automatically infers the associated OD information and uses it to produce the improved query plans.

4) *Inferring Order Dependencies.*
In Section III-D, 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 fuller advantage of these techniques.
   a) We discuss a general, efficient (polynomial) inference procedure which we have implemented and which is sound and complete over a *natural* domain. We define a database to be *natural* if given order properties over its attributes can be guaranteed. (All real-world domains we have encountered have these properties, and thus are *natural*.)
   b) We present two inference algorithms and show where and how they are invoked in the optimizer, and the advantages they can confer: *Reduce Order** which puts ODs into a canonical form for matching against *interesting orders*; and *Homogenize Order** which discovers equivalent columns, order-wise.

In Section IV, 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 on ten-GB database. (The average benefit on a one-TB database is >50%)

In Section V, 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 VI, we outline next steps for this work, and conclude.

II. Background

We adopt the notational conventions in Table I. We are interested in *lexicographical ordering*, or *nested sort*, as is provided by SQL's order-by directive. Given order by $\mathsf{A}_0, \ldots, \mathsf{A}_{n-1}$ 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, $\mathbf{X} = [\mathsf{A} \mid \mathbf{Z}]$, 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* $s_{\mathsf{A}} < t_{\mathsf{A}}$ *or* ($s_{\mathsf{A}} = t_{\mathsf{A}}$ 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 $\mathsf{A}_0, \ldots, \mathsf{A}_{n-1}$ 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.[2]

TABLE I
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.
  - $t_{\mathcal{X}}$ denotes the *projection* of tuple t on $\mathcal{X}$.
- **Sets**
  - calligraphic letters denote $\mathcal{X}$, $\mathcal{Y}$, and $\mathcal{Z}$ represent *sets* of attributes.
- **Lists**
  - bold letters represent *lists* of attributes: **X**, **Y** and **Z**. Note list **X** could be the empty list, $[\,]$.
  - square brackets denote an explicit list: $[\mathsf{A}, \mathsf{B}, \mathsf{C}]$.
  - $[\mathsf{A} \mid \mathbf{T}]$ denotes that A is the *head* of the list, and **T** is the *tail* of the list (the remaining list when the first element is removed).

TABLE II
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 $[\mathsf{A}_0, \ldots, \mathsf{A}_{m-1}] \mapsto [\mathsf{B}_0, \ldots, \mathsf{B}_{n-1}]$. A table—with attributes $\mathsf{A}_0, \ldots, \mathsf{A}_{m-1}$ and $\mathsf{B}_0, \ldots, \mathsf{B}_{n-1}$—*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 $\mathbf{r}$, $\mathbf{X} = [\mathsf{A}_0, \ldots, \mathsf{A}_{m-1}]$, $\mathbf{Y} = [\mathsf{B}_0, \ldots, \mathsf{B}_{n-1}]$, then $\forall s, t \in \mathbf{r}.\ s \preceq_{\mathbf{X}} t \rightarrow s \preceq_{\mathbf{Y}} t$.

For $\mathbf{X} \mapsto \mathbf{Y}$, we say $\mathbf{X}$ *orders* $\mathbf{Y}$. $\mathbf{X} \leftrightarrow \mathbf{Y}$ *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

$[\mathsf{d\_date\_sk}] \mapsto [\mathsf{d\_year}, \mathsf{d\_month}, \mathsf{d\_day}]$ and
$[\mathsf{d\_year}, \mathsf{d\_month}, \mathsf{d\_day}] \mapsto [\mathsf{d\_date}]$

are satisfied in Table II. On the other hand, order dependencies

$[\mathsf{d\_year}, \mathsf{d\_month}] \mapsto [\mathsf{d\_date}]$ and
$[\mathsf{d\_date\_sk}] \mapsto [\mathsf{d\_year}, \mathsf{d\_day}, \mathsf{d\_month}]$

are *falsified* by Table II.

Date and *time* are richly supported in the SQL standards. The widely-used benchmark standard TPC-DS [2] contains 99 queries. Of these 99, 85 involve date operators and predicates and five involve time operators and predicates. If the idea

[2]The inference problem for ODs with mixed directives is more computationally complex. So this restriction is not "without loss of generality".

of order dependency were only applicable to date and time, they could confer great benefit on query optimization.[3] 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 description can be found in [2].) 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 TPC-DS 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 a 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 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 [5]. 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.

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 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.

[3]The concept behind ODs is not limited to the time domain, however! ODs occur in other domains arising from business semantics. (We show an example in Example 2 in Section III-A.) 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 also order related (as with ODs).

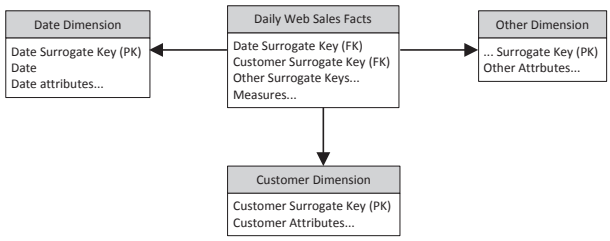

Fig. 1. Standard TPC-DS schema.

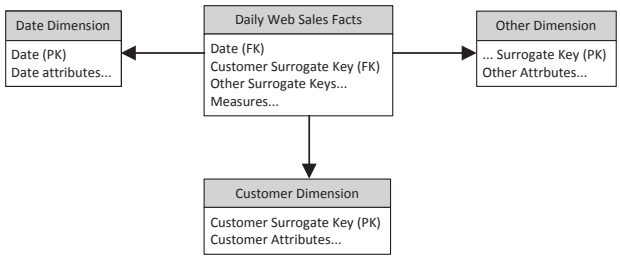

Fig. 2. Alternative TPC-DS schema with natural date key.

For our performance study (Section IV), 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 show different optimizations using ODs that can be accomplished. It also shows we achieve good performance improvements in both, so our techniques do not require specific schema designs.

III. Order Dependencies in Optimization

In Section III-A, we delve more in depth how order, and order dependencies, are used for optimization. Of course, for these techniques to add benefit, ODs must exist and be known. ODs can be *declared* on a database as *integrity constraints*. 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 gives a guarantee that the database will satisfy it. In Section III-B, we address this.

Even if a database has no declared ODs, OD-optimization techniques are still relevant. *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 $A = B$ in the where clause, then clearly the OD $[A] \leftrightarrow [B]$ 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 I-A. The optimizer must *detect* local ODs to use them. In Section III-C, 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 IV.

Even with the ODs declared for the database and the local ODs deduced in 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 $A = B$ in the where clause. If we also knew the declared OD $[A] \mapsto [Z]$, within the query's scope, OD $[B] \mapsto [Z]$ is also satisfied (by *transitivity* of ODs [6]). Therefore, the optimizer has a need to *infer* ODs from others. In Section III-D, we show when and where the optimizer would invoke OD-inference procedures, 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.

A. Optimizing with Order Dependencies

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. ODs can be used to great advantage in query processing, just as FDs have been [3].

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 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 extensions of window agreggation provide 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 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 [3].

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

Query 3: Template of the query.

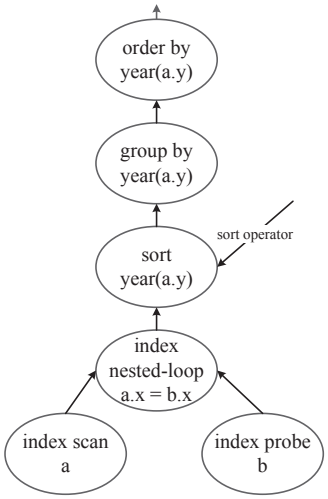

Fig. 3. Query access plan.

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 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.

The critical role of *interesting orders* was recognized quite early [7]. 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 subplans can produce are "interesting"; an order is not interesting if it is of no potential benefit to any other operator. This is a complex task. Many different orders could apply for an operation. For example, group by $A_0, \ldots, A_{n-1}$ can be accommodated on-the-fly by an input stream ordered by $[A_{p_0}, \ldots, A_{p_{n-1}}]$, 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 $[A_3, A_1, A_2, A_4]$, but not by $[A_3, A_1, A_4, A_2]$.

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 interesting order.

In [3], 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. We extend further on the techniques of [3] by also employing order dependencies to recognize more order equivalences.

B. Declaring Order Dependencies

In [4], 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 II, 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 (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 4, from the TPC-DS benchmark, requires that expensive join between the fact table web_sales and the dimension

```
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 4: With an expensive join.

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, which can be declared as a check constraint in DB2. Thus, 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 4 can then be simplified into the form shown in Query 5. (Details of when and how this rewrite can be performed in a general case appear in [4].)

```
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 5: Rewrite of Query 4.

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%.

C. 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] $\mapsto$ [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.

In this section, we describe our techniques using the monotonicity property. 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 [8] allows for rewrites in the case of queries with order-by. We then show the value of this technique when combined with indexing, including multi-dimensional clustering (MDC) in DB2. (MDC allows a table to be organized by an attribute into blocks of data, each with the same unique attribute values.) 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. 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 [8] for details.)

In [8], they show how to use this for predicate derivation, and this is why it was 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*. 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(1, _): 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.)
- concat(_): Returns the concatenation of two strings. (When 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 days(_) and hour(_); mathematical functions, such as log(_), ceil(_), and sqrt(_); and type conversions, such as int(_) and float(_).

Query 6 employs the substring function in its group-by. Recall Query 1 in Section I. 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 6: Substring with group-by.

Let there be an index on d_date in the date_dim table. In Query 7, the data are ordered by d_date converted to char, then *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 with a constant. This makes the OD [d_date] $\mapsto$ [to_char(d_date,'YYYYMMDD')||'12:00:00'], visible to the optimizer.

```
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 7: With string conversion and concatenation.

Query 8 can be effectively rewritten by the optimizer to the form in Query 9. An evaluation of the rewritten query then uses the index on d_date. The two constants 1998 and 2002 are used in Query 8 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 of date('1998-01-01') and date('1998-01-01'). Then, the optimizer uses index on d_date in the query plan.

```
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 8: With the predicate *year*.

Query 10 is similar to Query 6, 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

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

Query 9: Rewrite of Query 8.

could not be. It illustrates our OD techniques the same, though, in either case.)

```
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 10: Substring variation with order-by.

Query 11 is an OLAP query that uses a *partition-by* clause. The query plan can employ the index on d_date, given the optimizer detects via the monotonicity detection algorithm that the OD

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

follows, which effectively partitions by year concatenated with month.

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

Query 11: OLAP.

In Query 12 with a case expression, the monotonicity detection algorithm is also triggered; it detects that

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

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

```
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 12: With a case expression.

TABLE III
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

D. 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 this *natural* property of domains. 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. We present two algorithms, *Reduce Order** and *Homogenize Order**, that extend two algorithms in DB2 for matching interesting orders [3] further to accommodate ODs.

As discussed earlier, order dependencies are not limited to date and time. They commonly arise in many other domains.

Example 2: (Taxes) Consider table taxes in Table III, 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

$[\mathsf{salary}] \mapsto [\mathsf{taxes}]$,
$[\mathsf{salary}] \mapsto [\mathsf{percent}]$, and
$[\mathsf{salary}] \mapsto [\mathsf{group}, \mathsf{subgroup}]$.

It logically follows from these ODs that

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

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

Let the table taxes in Table III have a clustered index on salary. A query with order by taxes, percentage, group, subgroup given the three ODs as declared in Example 2 could then be evaluated using the index on salary, as the inference procedure could infer that

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

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

In Section III-B, we had assumed that $[\mathsf{date\_sk}] \mapsto [\mathsf{date}]$.

TABLE IV
SHOWING LACK OF TRANSITIVITY.

A	B	C
0	0	1
1	0	0

was declared. Instead, however, we may have had the following ODs:

$[\mathsf{date\_sk}] \mapsto [\mathsf{year}, \mathsf{month}, \mathsf{day}]$, and

$[\mathsf{year}, \mathsf{month}, \mathsf{day}] \mapsto [\mathsf{d\_date}]$.

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

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

We present an efficient inference procedure for testing logical implication of ODs which is *sound* and *complete* over *natural domains*. The details of the proofs can be found in [10].[4]

Definition 3: (order compatible) Two lists $\mathbf{X}$ and $\mathbf{Y}$ are *order compatible*, denoted as $\mathbf{X} \sim \mathbf{Y}$, *iff* $\mathbf{XY} \leftrightarrow \mathbf{YX}$.

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

Example 3: (Order compatibility is not transitive.) Assume $\mathcal{M} = \{\mathsf{A} \sim \mathsf{B}, \mathsf{B} \sim \mathsf{C}\}$. The table in Table IV satisfies the set of ODs $\mathcal{M}$. It falsifies $\mathsf{A} \sim \mathsf{C}$, however. This demonstrates that the order-compatibility relation is *not* transitive.

If we restrict our domain (the database) to have a property that guarantees a limited form of transitivity over *order-compatibility*, then we can make an efficient inference procedure for ODs.[5] The property we prescribe is *transitivity of order compatibility over single attributes*. We call a domain *natural* if it satisfies this property. Real-world domains do.

Definition 4: (Natural Domains) Call a database *natural iff*, for any three attributes A, B, and C from any given table, if the database satisfies $\mathsf{A} \sim \mathsf{B}$ and $\mathsf{B} \sim \mathsf{C}$, then it also satisfies $\mathsf{A} \sim \mathsf{C}$.

All of the schemas and domains we have explored, TPC-DS, and used in examples in this paper, are *natural*, by Definition 4. 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 5: ($\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$) The problem of *testing logical implication* for ODs is, given a set of ODs $\mathcal{M}$ and a OD $\mathbf{X} \mapsto \mathbf{Y}$, to decide whether $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$.

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) [10]. 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. We show that this can be verified in linear time. 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 4.) These parts combine to complete the proof of soundness and completeness of our inference procedure for ODs over natural domains.

Theorem 1: (correctness) [10] The algorithm for testing logical implication $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ for ODs is correct over natural domains.

Theorem 2: (complexity) [10] Testing logical implication for ODs over natural domains whether $\mathcal{M} \models \mathbf{X} \mapsto \mathbf{Y}$ is solvable in polynomial time, $O(|\mathbf{X}||\mathbf{Y}||\mathcal{M}|)$.

In [9], an axiomatization for order dependencies (defined as we have in this paper) over a restricted domain is presented. The author calls these *temporal functional dependencies* (TFDs), focusing on the time domain, and he provides 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 to us. It effectively restricts one to ODs of the form with just a single attribute on the right-hand side (e.g., $\mathbf{X} \mapsto [\mathsf{A}]$). In many of our examples, as in Examples 1 and 2, 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.

In [3], 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). Their rewrites rely on FD information available to the optimizer, but do not use *order dependencies*, which are the subject of our work.

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. We extend this algorithm—call it *Reduce Order**—by iterating through the list, additionally checking whether the *list* without the attribute being currently considered *orders* the full list. If so, the attribute can be dropped from the current list.

With this, we can optimize queries such as in Query 2 in

[4]The OD-inference problem over general domains is harder. In [10], we prove that the testing logical implication for ODs in general is co-NP-complete.

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

Algorithm 1 Reduce Order*

Input:
A set of ODs $\mathcal{M}$ and
order specification $\mathbf{O} = [\mathsf{O}_0, \mathsf{O}_1, ..., \mathsf{O}_{n-1}]$.
Output:
The reduced version of $\mathbf{O}$.

1: Rewrite $\mathbf{O}$ in terms of each column's equivalence class head.
2: //Scan $\mathbf{O}$ backwards
3: **for** $i \leftarrow n-1$ **to** 0 **do**
4: Let $\mathsf{B} = \{\mathsf{O}_0, ..., \mathsf{O}_{i-1}\}$
5: **if** $\mathsf{B} \rightarrow \mathsf{O}_i$ **then**
6: Remove O_i from $\mathbf{O}$
7: **else if** $[\mathsf{O}_0, ..., \mathsf{O}_{i-1}, \mathsf{O}_{i+1}, ..., \mathsf{O}_{n-1}] \mapsto [\mathsf{O}_0, ..., \mathsf{O}_{n-1}]$ **then**
8: Remove O_i from $\mathbf{O}$
9: **return** $\mathbf{O}$

Section I. We infer [d_month] $\mapsto$ [d_quarter] 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.

The algorithm *Reduce Order** is correct because removing O_i from the list using a FD $\mathsf{B} \rightarrow \mathsf{O}_i$ is part of *Reduce Order* algorithm described in [3]. Given an order dependency $\mathbf{X} \mapsto \mathbf{Y}$, the clause order by $\mathbf{Y}$, can be rewritten with order by $\mathbf{X}$, as strengthening the order-by conditions is allowed (as described in Section II). (In this case, $\mathbf{X} = [\mathsf{O}_0, \mathsf{O}_{i-1}, \mathsf{O}_{i+1}, \ldots, \mathsf{O}_{n-1}]$ and $\mathbf{Y} = [\mathsf{O}_0, \ldots, \mathsf{O}_{n-1}]$.)

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 reduced $\mathbf{O}$, an index can be matched, eliminating the need for a sort operator.

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

Interestingly, an order specification may become empty, as we are able to express it with order dependencies. For instance, if there is a predicate in the where clause $\mathsf{A} = 150$, which means that A is a *constant* in the scope of the query, this predicate yields $[\,] \mapsto \mathsf{A}$. Given $\mathbf{O} = [\mathsf{A}]$, $\mathbf{O}$ may reduce to empty.

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 [3]. It uses *equivalent classes* to substitute columns in an interesting order $\mathbf{O}$. We extend the algorithm as *Homogenize Order** to account for order dependencies.

The algorithm Homogenize Order* is correct because, given an OD $\mathsf{A} \mapsto \mathsf{B}$, if the data are ordered by A, they are also ordered by B (and vice versa), so eventually A can be substituted by B in an interesting order $\mathbf{O}$.

TABLE V
TABLE EMPLOYEE_SALARY.

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

Algorithm 2 Homogenize Order*

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

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

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 V.

Example 4: (Human Resource) A table employee_salary has the following attributes.

- id: employee identifier.
- position: corporate title.
- grade: employee evaluation.
- hire_date: date hired.
- salary: employee's salary.
- years: years of service.

Let there be a user-defined function *ranking* which takes as input the position of the an employee, and returns a number from 1 to 100. The higher the position, the higher the ranking. (For example, for the secretary, the function returns 30, while for the director, it returns 90.)

Salary rises as ranking, years of service, and grade rise, in that lexicographical order. That is,

$$[\mathsf{ranking(position)}, \mathsf{years}, \mathsf{grade}] \mapsto [\mathsf{salary}].$$

Moreover, the date when the person was hired is monotonic with respect to the identifier:

$$[\mathsf{ranking(id)}] \mapsto [\mathsf{date\_hire}].$$

Assume these order dependencies are declared as check constraints. The first constraint which expresses that

$$[\mathsf{ranking(position)}, \mathsf{years}, \mathsf{grade}] \mapsto [\mathsf{salary}]$$

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

$$[\text{ranking(id)}] \mapsto [\text{date\_hire}]$$

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.

IV. Experiments

We have implemented and tested in DB2 V10 the query rewrites described in Section III. 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 7 (Q1′ and Q7′, respectively), modified to match the alternative schema, and Query 11.

The experiments were performed on a performance testing machine with 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.

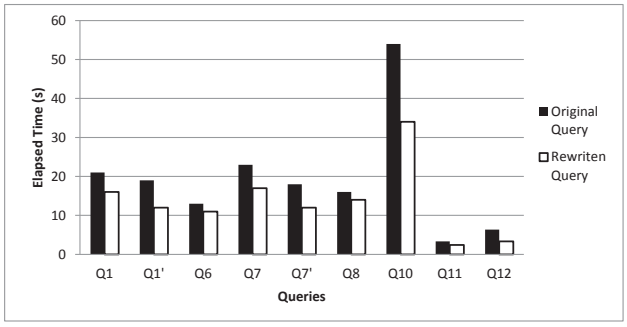

Fig. 4. Performance results.

Figure 4 shows the execution times for the nine queries modified from the TPC-DS benchmark and expressing real world IBM customers 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.

IBM was reluctant to permit specific results obtained on a one-TB TPC-DS database to be included at the time this paper was being written. We can report that on the one-TB database, the average performance benefit was >50% using our techniques for the nine queries. This indicates our techniques scale. So we are seeing performance improvements even increase as the the database scales. This was anticipated; these 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.

V. 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 has been recognized very early on. Right from the start, the query optimizer of System R [7] paid particular attention to *interesting orders* by keeping track of all such ordered sets throughout the process of query optimization. [11] and [12] explored the use of sorted sets for executing nested queries. The importance of sorted sets has prompted the researchers to look *beyond* the sets that have been explicitly generated. Thus, [8] shows how to discover sorted sets created as generated columns via algebraic expressions. (In DB2, a generated column is a column that can be computed from other columns in a table.) For example, if column A is sorted, so is the generated column G defined as G = A/100 + A - 3 (that is, A $\rightsquigarrow$ G). In [4] 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 [13]. Our concept of ODs is equivalent to *order-preserving mapping* 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.

Order dependencies were introduced for the first time in the context of database systems in [14]. 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 order over the values of *each* attribute of $\mathcal{X}$ implies an order over the values of *each* attribute of $\mathcal{Y}$. (For simplicity, we use the arrow $\rightsquigarrow$ for different type of orders. 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 [9] aptly described as point-wise versus lexicographical order dependency. 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 [11] 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 [15].

Dependencies defined over lexicographically ordered domains were introduced in [9] under the name *lexicographically ordered functional dependencies*. Two other papers [16] and [17] by the same author develop a theory behind both lexicographical as well as point-wise dependencies. (The latter were simpler than dependencies defined in [14].) 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 [6] we presented a sound and complete axiomatization for ODs. Polarized order dependencies (PODS) (a mix of asc and desc) were introduced in [18].

VI. 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 removes a potentially expensive sort operator.

The techniques described in this paper, although implemented in IBM DB2 V10, are general enough to be used in any other query optimizer. These techniques should apply to a wide range of BI queries. Our experiments show viability of the proposed solutions. Nine queries described in the paper benefited, with an average gain of 30% on ten-GB database and more than 50% on one-TB database.

In future work, we plan to pursue two lines of research: first, in further practical applications of ODs; and second, in the theoretical domain.

- 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 [3], while inclusion dependencies are employed to remove redundant joins over primary and foreign keys [19]. We would further investigate how order dependencies can be useful in query optimization. This includes monotonicity in case expressions and optimization of the queries such as Query 13, where there is an order dependency between 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 order dependencies can also be identified in *geo-spatial* dimensions of a data warehouses. Similar optimization techniques which we have described in the paper can be applied there, too.
- We are trying to improve the integration of our 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 even more beyond what we have already gained with our query-rewrite techniques.
- We plan to work on axiomatization for polarized order dependencies [6] which allow the mix of ascending and descending orders. Such axiomatization might provide insight into how polarized order dependencies behave, and provide input for useful query rewrites.

Acknowledgments

IBM, the *IBM logo*, and *ibm.com* are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web as "Copyright and Trademark Information" at *http://www.ibm.com/legal/copytrade.shtml*.

TPC-DS is a trademark of The Transaction Processing Performance Council.

References

[1] W. Armstrong, "Dependency structures of data base relationships," in *Proceedings of the IFIP Congress, Stockholm, North-Holland 580-583*, 1979.

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

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

[4] 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.

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

[6] J. Szlichta, P. Godfrey, and J. Gryz, "Fundamentals of Order Dependencies," in *VLDB*, 2012.

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

[8] 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.

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

[10] J. Szlichta, P. Godfrey, and J. Gryz, "The Complexity of Order Dependency Inference," in *Technical Report, http://www.cse.yorku.ca/techreports/*, 2012.

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

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

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

[14] S. Ginsburg and R. Hull, "Ordered Attribute Domains in the Relational Model," in *XP2 Workshop on Relational Database Theory*, 1981.

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

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

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

[18] J. Szlichta, P. Godfrey, and J. Gryz, "Chasing Polarized Order Dependencies," in *The Alberto Mendelzon International Workshop on Foundations of Data Management*, 2012.

[19] 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, 1-298*, 1999.