1,000 Tables Under the From

Nicolas Dieu, Adrian Dragusanu, Françoise Fabret, François Llirbat, Eric Simon

SAP Business Objects
Levallois, France
firstname.lastname@sap.com

ABSTRACT

The goal of operational Business Intelligence (BI) is to help organizations improve the efficiency of their business by giving every "operational worker" insights needed to make better operational decisions, and aligning day-to-day operations with strategic goals. Operational BI reporting contributes to this goal by embedding analytics and reporting information into workflow applications so that the business user has all required information (contextual and business data) in order to make good decisions. EII systems facilitate the construction of operational BI reports by enabling the creation and querying of customized virtual database schemas over a set of distributed and heterogeneous data sources with a low TCO. Queries over these virtual databases feed the operational BI reports. We describe the characteristics of operational BI reporting applications and show that they increase the complexity of the source to target mapping defined between source data and virtual databases. We show that this complexity yields the execution of "mega queries", i.e., queries with possible a 1,000 tables in their FROM clause. We present some key optimization methods that have been successfully implemented in SAP Business Objects Data Federator system to deal with mega queries.

1. OPERATIONAL BI APPLICATIONS

In this section, we describe the characteristics of operational BI applications and the requirements for operational BI reporting.

1.1 Operational BI characteristics

The goal of operational BI is to help organizations improve the efficiency of their business by:

- giving every "operational worker" insights needed to make better operational decisions
- aligning day-to-day operations with strategic goals

VLDB '09, August 24-28, 2009, Lyon, France.

While traditional BI focuses on tactical and strategic types of decisions to the intention of middle managers, information workers, executives, and senior managers, operational BI focuses on operational types of decisions for supervisors and operational workers.

Operational Workers (aka Structured Task Workers) are typically bank clerks, call centre operators, nurses and people in supervisor roles (shop managers, bank managers, etc.)

Operational decisions are decisions made by business users in day-to-day operations (e.g., handle a call in a call center; supervise the action of multiple agents, validate a purchase order; supervise a supply chain; etc). There, the goal is to give these users the most trusted and relevant information to improve their decisions. Studies show that most operational decisions are made by operational workers (70% of the decisions they make) while it represents less than 20% of the decisions made by middle and above managers.

Operational decisions have important characteristics. First, they are highly *contextual*. They are typically part of a business process (automated or not via business workflows) and they depend on rich contextual information such as the context associated with a particular processing step in the workflow, and the strategic goals to which the operational decisions must contribute (e.g., a forecast value of some KPI).

Second, they are highly *repeatable*. The same type of decisions is made quite often, usually with the same premise but with a different context. For instance, the caller, her location, the reason for the call, etc are different although the type of action needed is a well registered procedure.

Third, operational decisions have a *small action distance*. Decisions are made by those close to execution and the effect of action taken must be seen quickly by the operator who takes it but also by other operators because it may change the context in which subsequent decisions will be taken.

Operational BI applications take place in a decision loop (also called lifecycle of business execution) with the following consecutive phases:

- *Awareness*: handles process monitoring and operational intelligence
- Insight: handles business intelligence and data mining

- Strategy: handles planning and budgeting, scorecards and goals, process modeling and optimization
- *Execution*: handles process execution through enterprise applications and workflows

Operational decisions cover the phases of execution and awareness. Operational BI creates the interface between these two phases and the two other phases.

1.2 Operational BI reporting requirements

Referring to the above phases of the decision loop, the area of focus of operational BI reporting is to link strategy to execution (i.e., use business goals business data to drive process execution, embed analytics and reporting information into workflow applications), and link awareness to insight (i.e., show impact of operational decisions, link operational and process metrics to business warehouse data).

More specifically, the purpose of operational BI reporting is to present the business user with all required information (contextual and business) in order to make good decisions. *Business data* are dependent on the process instance (e.g., the caller or the purchase order) while *contextual data* reflect a more global situation (e.g., global KPI, business warehouse data). For instance, in a workflow for validation of a purchase order (especially in times of strong control of spending …), a typical operational BI report requires business data (e.g. the data associated with the submitted purchase order), and contextual data (what needs to be verified; what is the strategic goal) that may depend on the type of order.

Both business and contextual data reside in many different types of sources. Business data are typically in flat files, OLTP systems, Enterprise Resource Planning (ERP) applications, or streams. Contextual data are usually in spreadsheets, Data Warehouses (DW) or Data Marts (DM), database servers or web services. The number of data sources for a given customer can also be large. Studies report that it is greater than 20 in 29% of the cases for very large corporations and 15% of the cases for large corporations, and varies between 5 and 20 in at least 34% of the cases.

Highly repeatable decisions are captured through predefined reports (also called enterprise reports) that use BI tools to enable their fast deployment. These reports are based on highly parameterized queries (by means of prompted values to the business user) to the underlying data sources that contain business and contextual data. The fact that reports are predefined means that it is possible to know (or learn) all the queries that will be issued to feed the reports, a notable fact when we will come to query optimization later in this paper.

Regarding small action distance, operational BI reports must reflect data with high accuracy and timeliness. Studies report timely data requirements of: intraday (46%), intrahour (25%), intraminute (7%). This typically concerns business data and raises the challenge of providing direct access to these data without staging them in intermediate data repositories.

1.3 Operational BI reporting architecture

A typical operational BI architecture deployed by SAP Business Objects at customer sites distinguishes three main logical layers:

- a Data Source layer that comprises all the required data sources needed by operational BI reporting
- a Semantic Layer, that provides a layer of virtual "business objects", called a "universe" in Business Objects terminology, that can be easily composed to build reports
- a Consumption Layer that comprises multiple client tools such as reporting, dashboards, planning, data explorer, etc.

The Semantic Layer and the Consumption Layer are packaged into a single Business Suite. The semantic layer itself offers the following layered functionalities:

- a *data access* functionality standing at the bottom level, which provides a unified interface to retrieve metadata and query the set of distributed and heterogeneous data sources
- a *data foundation* functionality standing at the middle level, which enables the creation of multiple entity-relationship models from the metadata exposed by the data access layer
- a *business object* functionality, which enables the creation of so-called "business objects" (such as measures, dimensions, details, etc) that can be freely composed to create ad-hoc (BO) user queries consumable by the consumption tools (e.g., reporting or dashboarding tools).

In order to facilitate the fast deployment of operational BI reports, the Semantic Layer must provide fast means to create data models from multiple data sources without having for instance to require the development of intermediate data marts. Enterprise Information Integration (EII) systems [5], also called mediation systems, play a key role in the architecture of the Semantic Layer because they enable the creation and querying of virtual databases, henceforth called *target schemas*, over a set of distributed and heterogeneous data sources with a low TCO (Total Cost of Ownership). Within SAP Business Objects offering, the EII system is named Data Federator.

2. EII SYSTEMS FOR OPERATIONAL BI

In this section, we focus on the requirements conveyed by operational BI applications on EII systems.

2.1 Preliminaries on EII systems

An EII system is a middleware that includes wrappers and a query engine. Wrappers expose the metadata relational schema of the data sources called source tables. A wrapper describes also the capabilities of the underlying data source in terms of SQL query processing over the source tables. This model must be sufficiently powerful to capture the sometimes subtle limitations of SQL support in a data source (typically for systems like Progress and SAS, for instance). The Query Engine (QE) provides a SQL interface to express multi-source queries referring to source tables or *target tables* (i.e., tables belonging to target schemas).

At the time of creation of a target schema, a *source to target mapping* is created between the source tables and the target tables

belonging to a given target schema. An arbitrary number of intermediate target tables can be used within this mapping to facilitate its definition.

At the time of execution of a multi-source SQL query by the Query Engine, the query is first unfolded using the mapping definition of the target tables and then decomposed into a set of *subqueries*, that are processed by the wrappers, and an *assembly query* that is processed by the QE using the results returned by the subqueries. This decomposition is called a *query processing plan*.

2.2 Target Schemas

It is important to keep in mind that a target schema is entirely driven by the reporting requirements that it must serve, so its design follows a top-down approach. Thus, the target schema is created before the mapping actually occurs.

In operational BI reporting applications, target schemas often have a "star"-like configuration, which distinguish *fact tables* from *dimension tables*. Fact tables can either be measures (in the traditional sense of multi-dimensional data modeling) such as sales-unit, or sales-dollar, or entity data that are described using multiple dimensions, such as a customer or an archive box. Indeed, customer data can be a subject of analysis that is described using different dimensions (location, organization, market position,…). So, a customer is not necessarily a dimension, this depends on the viewpoint of the business process to which the operational BI model is associated with. Finally, dimension tables are frequently de-normalized because these tables are virtual read-only tables and it makes no sense from a reporting point of view to decompose them in smaller tables.

2.3 Complex Source to Target Mappings

In our experience, we have observed three main sources of complexity for target schemas in operational BI reporting: (i) source data fragmentation, (ii) complex logic for contextual information, and (iii) data cleaning operations.

First, source data are frequently fragmented even if the reasons vary from one customer to the other. Consider for instance a Customer dimension table with attributes such as ID, name, address, home-phone, nation-name, etc. Vertical partitioning may occur because the attributes of a customer originate from multiple source tables (e.g., detailed data coming from Cust-support). Then horizontal partitioning may occur because the source tables have been partitioned into multiple databases. For instance, Customer data can be co-located with orders data while customer support data can be partitioned into regional databases. In this example, this means that the mapping definition of the Customer table will include a first level of union operations to address horizontal partitioning and then a level of join operations (join, outer joins) to address vertical partitioning.

Master databases may reduce the complexity created by fragmentation because centralized consolidated tables (the master tables) can be leveraged. This occurs typically for dimension data such as a Customer table. However, it does not eliminate data fragmentation. First, not all data about a customer are kept in a customer master table and additional data can be aggregated on demand. Second, lookup tables need sometimes to be built to keep the relationships between the master table and the "original" source tables from which the master table has been built. Indeed, applications may continue using the original tables instead of the consolidated master tables. These lookup tables are usually co-located with the original source tables. Third, fact tables also rely on fragmented source data, as our previous example showed.

A second source of complexity is the complex logic that can be associated with a fact table, especially when it represents contextual information. Indeed, contextual information may consist of business indicators that are not readily available in the underlying data sources but that require some computation. Consider for instance, a fact table called Box_Destruction that describes all the archive boxes that are eligible for destruction. The data sources store information about archive boxes (for short, called boxes), their customer, the organization within the customer, etc. However, the notion of a box eligible to destruction is a business notion that needs to be computed. In this example, a box must be preserved if it has not passed its expiration date or:

Case 1: it is on hold by the company or

Case 3: it is on hold by the parent company or

Case 3: it is on hold by a division or

Case 3: it is on hold by a department

Each case yields an intermediate target table involving joins or left outer joins and selections, and the union of these target tables forms all the "box on hold" cases in a Box_Hold table. Finally, the Box_Destruction tuples are obtained by taking the boxes that are "not in" the Box_Hold table, which again could be implemented by a left-outer join operation. So, the computation of the Box_Destruction table requires an interleaved combination of joins, left outer joins, selections and unions. In addition, if box data are fragmented using horizontal partitioning. This computation must be done for each partition and then a final union will yield the result.

Finally, a third source of complexity is the necessity to perform some data cleaning operations to enable the relationship between data from different sources and to guarantee the accuracy of the target data. The cleaning issues addressed by the EII mapping must be translatable into tuple-transformations, which means that the transformation operations can be applied to each tuple individually. Hence, issues such as duplicate elimination are not considered here and are usually addressed elsewhere (either the source data are cleaned before hand or a clean database has been created using an ETL tool). The impact on source to target mappings is the presence of multiple evaluation expressions (aka functional expressions) including *case-when* expressions and string manipulation functions.

3. THE CHALLENGE OF MEGA QUERIES

We first introduce mega queries and explain how they may occur. We then introduce the notion of data processing set as our main optimization goal and explain its rationale.

3.1 Mega Queries

Mega queries are SQL queries which reference a large number of source tables (several hundreds to more than a thousands) in their

FROM clause. Such queries can be issued by operational BI reports as a result of the complexity of the source to target mappings used to define the underlying target schemas. This however does not equate to a comparable number of joins since horizontal partitioning also creates many union operations.

Building upon our previous examples taken from a customer use case, there are more than 300 databases of up to 100 GB each of which containing source data about archive boxes. There is a central master database that holds dimension data such as customer and customer's organization. All the data about the boxes and their contents is horizontally partitioned over about 80 different databases within each region. The data about the warehouses is also horizontally partitioned over 10 different databases.

In a business process workflow, an operational BI report requires business data that describe detailed customer information associated with boxes eligible to destruction and the warehouses where the boxes are kept. There is also the possibility of drilling down into a particular archive box to see the list of files stored.

The target schema exposed for this operational BI report contains tables such as Customer, Box_Destruction, Warehouse, Files. The query used to report information on boxes eligible for destruction is merely expressed by joining target tables Customer, Box_Destruction, and Warehouse. Customer has a simple mapping from the master database that mixes customer and organization data. Box_Destruction has a complex mapping sketched in the previous section. Finally, Warehouse also has a simple mapping. However, after unfolding, the query references over 1,000 source tables from more than 80 different databases. Indeed, each case for defining Box_Hold involves 3 to 5 joins, so a total of about 15 tables for each partition. This gets multiplied by the number of roughly 80 partitions, yielding a total of 1,200 tables. Then, a total of roughly 15 tables is added for the warehouse and master data. The filters that make this query narrow are a filter on a customer ID and a region.

3.2 Data Processing Set

As mentioned before, in an EII system, a query processing plan P (or query processing tree) for a query Q is composed of the following macro-components:

- subqueries from the data sources, noted SQ, sometimes called "data providers" (we shall denote $[SQ]_S$ a subquery SQ on source S),
- an assembly query AQ run in the EII Query Engine

We introduce the notion of *data processing set* for a query plan P of query Q, noted dps(P(Q)), as the number of rows that are read from the data sources by the EII Query Engine in order to complete the processing of AQ.

First, we define the data processing set of a subquery SQi, noted dps(P(Q), SQi), as the number of rows returned by SQi in query plan P(Q). If N is the number of subqueries in P(Q), we have:

$$dps(P(Q)) = \sum_{i=1}^{N} dps(P(Q), SQi)$$

3.3 Narrow Queries

We have observed that many queries for operational BI reporting are "narrow" queries, which means that their data processing set can be quite small although the source tables that they reference can be very large. For instance, some fact tables issued from large data warehouses such as Teradata or Netezza databases can be extremely large.

The reason for narrow queries is that highly repeatable decisions in workflow processes are typically captured by highly parameterized queries in the operational BI reports. Usually, the query parameters represent filters on dimension tables such as a customer ID value, a region, a quarter or a range of dates, a product ID, etc. One of these filters is usually quite selective because it corresponds to the specific context of a particular instance of a business process. For instance, the customer ID of the caller in a self-service system will be specific although the region where the customer belongs is not. In the customer use case above dealing with archive boxes, 60% of the mega queries reference more than 300 tables, while 12% reference more than 1,000 tables. However, 83% of the mega queries were narrow queries that could be optimized and processed in a range of 20 s to 2 minutes.

Narrow operational BI queries are allowed to query production systems (e.g., OLTP or ERP systems) for accessing their business data because they generally issue small read transactions. Thus, they are not causing problems in workload balancing.

In addition, a notable proportion of the queries return a small result set because operational workers need condensed and canned information to take decisions. Progressively, we also see mobile operational workers which use mobile devices to display their analytics and BI reports using specific interactive graphical widgets.

3.4 Optimization Goal and Challenge

Our optimization goal is to find a query plan P for Q that has the smallest dps(P(Q)). This goal does not guarantee that we produce the cheapest query processing plan but it provides a good approximation for two main reasons. Firstly, data transfer turns out to be a dominant cost factor in query processing in an EII, so it is imperative to limit it. This is particularly true for operational BI queries that may access very large tables. Secondly, in all cases where there exists a query plan with a small data processing set (for narrow queries), it is important to find such a plan because then the assembly query can be run extremely fast without worrying too much about further ordering of operations. On the contrary, missing a plan with a small data processing set can be dramatic in terms of performance.

A main challenge for a query optimizer is the size of the search space created by a mega query. Conventional techniques using rule–based optimization, whereby rewriting rules are recursively considered for application starting from the root node of a query plan, do not apply because they do not scale both in time and space. Similarly, dynamic programming techniques for join re-ordering do not scale on mega queries although they may work on smaller (partial) plans. In our previous implementation based on conventional query optimization design principles, our optimizer would take hours to optimize a query when it was not eventually running out of memory. Over simplifying the optimization rules

would be insufficient because then we would miss our optimization goal.

4. OPTIMIZE THE OPTIMIZER!

We first needed to radically change the way our query optimizer works in order to address the complexity of mega queries. In this section, we present our two main design decisions, that is, the decomposition of the optimization process into sequential stages and the design of a new compact representation of a query plan that significantly reduces the cost of query optimization with respect to our optimization goal. Other important decisions such as when and how to collect statistics and how to propagate them are left out of this paper.

4.1 Decompose optimization in stages

We first decomposed the optimization into successive "stages" where each stage focuses on a particular type of optimization. Our optimization stages are processed sequentially and only once. They are designed with two objectives: (i) guarantee a low worst case time complexity (linear or polynomial), and (ii) low memory consumption, in particular favor "in place" transformations to avoid the cloning of plans, which is very costly in memory space. In this paper, we only discuss the following stages:

- Simplification and push-down of operations
- Searching for maximal subqueries
- Semi-join data provider reduction

Query simplification techniques perform transformations such as the removal of useless duplicate columns in DISTINCT or GROUP BY clauses, the removal of ORDER BY operators, the replacement of UNION DISTINCT by UNION ALL, the removal of useless columns/expressions that are not used /projected in the upper part of a query plan, the grouping of common expressions, etc. In particular, we make two assumptions:

- Useless LOJ are removed: if we have R LOJ S with an equality condition on S primary key columns and no column of S is used in the upper part of the plan the LOJ is removed.
- False LOJ are transformed into Joins: if we have R LOJ S followed by a null rejecting filter on a column of S then the LOJ is transformed into a Join.

The push down of operations is done by exploiting the standard commutativity and associativity properties of operators, in particular for filters, evaluations, aggregates and unions [2], [4], [8], [9]. Important properties to exploit during these transformations are the dependencies between variables, variable equivalence for transitive propagation, and behavior with respect to null values. These transformations are quite important because they prepare the work for the computation of maximal source subqueries.

4.2 Compact Query Plan

Our second decision to address the scalability problem of mega queries is to perform our optimizations on a compact representation of a query plan, called Compact Query Plan (CQP). We first introduce a couple of useful notions.

If Exp(x1, …xn) denotes a condition in conjunctive normal form with variables x1 to xn, then Exp is said to be *null rejecting* if for each i, $1 < i < n$, (xi = null) implies that Exp(x1, …xn) evaluates to false. Otherwise, Exp is said to be a *null accepting* condition. By extension, if an LOJ (resp. FOJ) has a condition that is null accepting, then the LOJ (resp. FOJ) is said to be null accepting.

Let Eval(x1, …xn) denote an evaluation operation with variables x1 to xn, then Eval is said to be a null propagating evaluation operation if there exists i, $1 < i < n$, such that (xi = null) implies that Eval(x1, …xn) evaluates to null. Finer sufficient conditions are used in practice but these details are left out of the scope of this paper.

A CQP distinguishes three types of nodes:

- Abstract node: contains a query plan restricted to the following operators: Cartesian product (C), Filter (F), Evaluation (E), Left Outer Join (LOJ)[1] and Full Outer Join (FOJ) that are null rejecting.
- Singleton node: contains one of the following operators: Aggregate Group by (A), FOJ that are null accepting (FOJnull), Order by (O), and set-oriented operations like Unions (U) and Minus (M).
- Source node: contains a subquery to a data source.

Before describing the construction of a CQP from a query plan, we introduce a few notions.

Let (N1, …, Nk) be a descending path in a query plan such that Nk is a source table or Nk is in {A, O, FOJnull, U, M}, and for each i, $1 \le i \le k-1$, (i) Ni is in {C, E, F} or (ii) Ni = LOJ and Ni+1 is the outer side of Ni. Then P = (N1, …, Nk-1) is said to be a *LOJ preserved path* in the query plan.

If (N1, …, Nk) is path such that P = (N1, …, Nk-1) is an LOJ preserved path and Nk is a singleton node then we associate a *variable node* with P that contains the set of output variables of node Nk.

We also introduce the notion of FOJ preserved path. Let P = (N1, …, Nk) be a descending path in a query plan such that N1 and Nk are FOJ, and for each i, $2 \le i \le k-1$, Ni is either (i) an FOJ or (ii) a null propagating evaluation node, or (iii) Ni is a null rejecting LOJ and Ni+1 is the outer side of Ni. Then P is said to be an *FOJ preserved path* in the query plan.

If P = (N1, …, Nk) is an FOJ preserved path and N, N' are the child singleton nodes of Nk, then we associate a *variable node* with P that contains the set of output variables of N and N'.

BUILD COMPACT QUERY PLAN

Input: an (original) query plan

Output: a compact query plan

Method: traverse the original query plan from its root in a depth first, left to right descending manner as follows:

Case 1: If an operator A, O, FOJnull, U or M is encountered in the query plan, then a singleton node N is formed in CQP. Its parent node (if it is not the root) in the CQP is the node of the CQP that contains the parent of N in the original query plan.

[1] Right outer joins are modeled using LOJ and joins are modeled using C and F.

Case 2: if an operator in {E, C, F, LOJ, FOJ} is encountered in the query plan it forms the root, called rootAN, of an abstract node AN which is built using the following rules:

1. if rootAN is not an FOJ, all maximal LOJ preserved paths[2] rooted at rootAN form an additional *block expression* of AN.
2. if rootAN is an FOJ, all maximal FOJ preserved paths[3] rooted at rootAN form an additional *FOJ block* of AN.
3. if N is an FOJ node that is a child of a node of any block expression of AN, all maximal FOJ preserved paths rooted at N form an additional FOJ block of AN.
4. if N is the non outer side node of an LOJ node of any block expression of AN and N is in {E, C, F, LOJ}, all maximal LOJ preserved paths rooted at N form an additional block expression of AN
5. if N is a child node of any block FOJ of AN and N is in {E, C, F, LOJ}, all maximal LOJ preserved paths rooted at N form an additional block expression of AN.
6. if N is a singleton node that is the non outer side of an LOJ node of a (expression or FOJ) block, then a variable node containing all the output variables of N is associated with the LOJ node.
7. if N is a singleton node and N is a child node of an FOJ block, then a variable node containing all the output variables of N is associated with the FOJ block.

The parent node (if it is not the root) of an abstract node AN is the node of the CQP that contains the parent node of rootAN in the original query plan.

Case 3: if a scan operator of a source table is encountered then a source node N is formed in the CQP. Its parent node in the CQP is the node of the CQP that contains the parent of N in the original query plan.

Finally, we remove from each (expression or FOJ) block the null rejecting LOJ operators (together with their associated variable node) and add them as new individual LOJ blocks in the abstract node.

END – BUILD COMPACT QUERY PLAN

Thus, each abstract node has the following constituents: a set of Block Expressions (BE) containing a set of paths, a set of FOJ blocks (BFOJ) also containing a set of paths, and a set of LOJ blocks. Rules 6 and 7 above add variable nodes to a block. Informally, the variable nodes of a block represent the variables that come from outside the abstract node that contains the block.

A block expression is internally represented by a *variable dependency graph*, a DAG that connects the sets of evaluation variables, filters, LOJ conditions, and variables of the variable nodes. Similarly, a variable dependency graph is used for FOJ blocks.

EXAMPLE 1: Applying BUILD COMPACT QUERY PLAN on the query plan of Figure 1, we first get a singleton node (Order By), then an abstract node AN consisting of one block expression B1 (via rule 1 above) and an FOJ block (via rule 3 above). Then we extract the two null rejecting LOJ to get blocks B2 and B4. Note that the LOJ node inside B1 is a null accepting LOJ. Finally, each source table will form a source node.

The variable nodes of B1 contain all the variables coming from T1 and T2 (via their preserved paths). The variables nodes of B3 contain all the variables coming from T3 and T4 (via JOJ preserved path) and the variables coming T6 (via rule 7 above) ¤

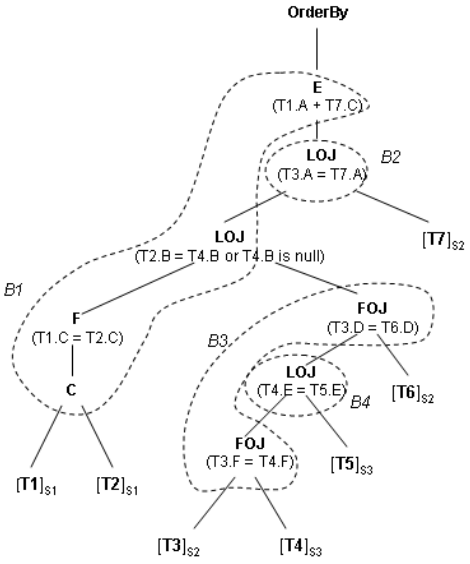

Figure 1: Abstract node in a query plan

In the example of the archive boxes given earlier, the CQP for the query that retrieves the boxes eligible to destruction has about 600 nodes due to the presence of many union nodes.

4.3 Execution Plan for an Abstract Node

Precedence relationships determine the ordering constraints under which the blocks of an abstract node can be executed. They are represented by *precedence links*, noted p-links, which describe variable propagation flows between nodes up to the rootAN of the abstract node. The definition of p-links derive from the standard properties of relational operators: associativity and commutativity of joins, associativity of FOJ, pseudo-associativity of LOJ and joins, asscoiativity of LOJ, pseudo-associativity of LOJ and FOJ.

Let AN be an abstract node, P-links are defined as follows:

- if N is the root node of an FOJ block B and N' is the parent node of N in the original query plan, then (i) if N' belongs to an FOJ or LOJ block[4] then there is a p-link B → N', otherwise (ii) there is a p-link B → B' where B' is the block expression containing N'.
- if N is the root node of a block expression B and N' is the parent node of N in the original query plan, then if N' belongs to an FOJ block B' then there is a p-link B → B'

[2] Including the variable nodes of the maximal paths.

[3] Including the variable nodes of the maximal paths.

[4] Necessarily on the non outer side of N'

The semantics of a p-link B1→ B2 is that B1 must be completely computed before B2 can consume the output variables of B1. Thus, a p-link creates a blocker in the pipelined evaluation of the nodes.

EXAMPLE 2: in the abstract node of Figure 1, we have only one p-link: B3→ B1 ¤

Let AN be an abstract node, and B1 and B2 be two (expression or FOJ) blocks of AN. The variables of B1 can be *propagated* to B2 if either (i) there is a p-link B1→ B2, or (ii) there exists an LOJ block B3 such that there is a p-link B1→ B3, and the preserved variables of B3 can be propagated to B2, or (iii) there exists a block B3 distinct from an LOJ block such that the variables of B1 can be propagated to B3 and the variables of B3 can be propagated to B2.

If B is a (expression or FOJ) block then the *output variables of B* is the union of the set of variables that can be propagated to B and the set of variables created in B (through evaluation nodes).

A *valid execution plan* AnP for an abstract node is a tree whose nodes are either block expressions of FOJ blocks. Supposing that all LOJ blocks are attached to a (expression of FOJ) block that provides all the preserved variables of the LOJ, AnP is defined as follows:

- Leaf node: a (expression or FOJ) block is a leaf node of a query plan AnP if all its (used) variables are contained in its variable nodes.
- Intermediate node: a (expression or FOJ) block B is a parent node of B1, …, Bp if all its (used) variables are contained either in its variable nodes or in the set of output of B1, …, Bp.

EXAMPLE 3: In the abstract node AN associated with the query plan of Figure 1, the only leaf node is B3. Then B1 is an intermediate node. Block B4 can only be attached to B3 while B2 can be either attached to B1 or B3. ¤

5. OPTIMIZATION TECHNIQUES

In this section, we present two main techniques that have proved to be the most effective to find a query plan with a small data processing set: create maximal source subqueries, and use semi-join reductions. We show how these techniques take advantage of a Compact Query Plan to handle the complexity of mega queries.

5.1 Maximal Source Subqueries in a CQP

There is an optimization stage devoted to the construction of maximal source subqueries. As said earlier, this stage follows a stage of simplification and push-down transformations whereby singleton nodes of a CQP have been pushed down.

5.1.1 Source capability model

A key feature for enabling the construction of maximal subqueries is the capability model of the data sources. The model distinguishes between two types of capabilities: (1) SQL-based capabilities, which indicate the SQL operations supported by the data source independently from the tables and columns to which the operations apply, and (2) metadata-based capabilities, which overwrite a SQL capability depending on the tables and columns to which the operations are applied. In particular SQL-based capabilities describe the evaluations (functional expressions) that can be computed by a data source. This is quite important because as said earlier, source to target mappings contain many evaluations.

5.1.2 Finding Maximal Subqueries

We assume that we have as many colors as there are data sources plus two special colors NONE and UNKNOWN, and we use the colors to mark the nodes of the CQP. Initially, all source nodes are colored with the color of their sources.

In what follows, we define a source subquery SQ to be analogous to an abstract node. So, it may consist of a set of blocks (expression, FOJ, LOJ), and a set of p-links.

CQP MAXIMAL SUBQUERY

Input: a CQP

Output: a new CQP in which abstract nodes and source nodes have been modified

Method: start from the source nodes of a CQP which are simple scans of source tables, and perform a bottom-up left to right traversal of a CQP.

Case 1: if a singleton node N is encountered:

1. If (i) its input nodes in the CQP all have a color C distinct from NONE, (ii) N satisfies the source capabilities of source C, and (iii) it is *profitable* to push N into C, then the tree rooted at N is moved to a source node of color C and the parent of N is examined
2. Otherwise, mark N with color NONE and stop tree traversal from N

Case 2: if an abstract node N is encountered then search for the maximal source subqueries that can be extracted from N, say SQ1, …, SQn.

1. if there is a single SQ1 of color C that is equivalent to N then the tree rooted at N is moved to a new source node of color C and the parent of N is examined.
2. otherwise, create *n* source nodes for the SQi subqueries, remove them from N, merge them with the subqueries associated with the source nodes input of N. Finally, mark N with a NONE color and stop tree traversal from N[5].

END – CQP MAXIMAL SUBQUERY

The profitability condition used in Case 1 is a heuristics that considers multiple cases, for instance for a union singleton node, which is not always worth pushing. Details are beyond the scope of this paper.

We now explain the search for maximal subqueries within an abstract node:

ABSTRACT NODE MAXIMAL SUBQUERY

[5] Subqueries can actually be "challenged" by marking them and continuing the bottom-up traversal of the CQP to see if further merges are possible, without affecting the linear complexity of the method (still a single traversal). Details are beyond the scope of this paper.

Input: an abstract node AN

Output: a set of source subqueries other than table scans, and a possibly new node AN' where subqueries have been removed from the expression or FOJ blocks, empty blocks are removed, and LOJ blocks have been removed.

Method: follows the following steps

Step 1: Initialization

For each (expression or FOJ) block that qualifies as a leaf node in a valid query plan, search for maximal subqueries SQ1, …, SQn. Using the same analysis as in Case 2 of the previous method, we end up with either a block of color C with a single associated subquery, or a block of color NONE with several associated subqueries. The output p-link of a block takes the color of its block.

Step 2: bottom-up traversal of p-links and marking of all nodes

Start from colored blocks and perform a bottom-up traversal of the graph of p-links until all blocks are either marked as NONE or UNKNOWN.

Case 1: B is a colored block, there is a link B → B' and the used variables of B' are contained in the union of its variable nodes and its propagated variables. Search for maximal subqueries SQ1, …, SQn of B':

1. if there is a single SQ1 of color C that is equivalent to B' then SQ1 is merged with the subqueries associated with the blocks that point to B', and B' is marked with color C.
2. otherwise, the SQi subqueries are merged with the subqueries associated with the blocks that point to B', and B' is marked with color NONE.

Case 2: B is a colored block, there is a link B → B' and there are used variables of B' that are not in the union of its variable nodes and its propagated variables. Search for maximal subqueries SQ1, …, SQn of B'. By assumption, there will be multiple subqueries SQi, (possibly with the same color C). Then, B' is marked as UNKNOWN.

Case 3: B is a colored block, there is a link B → B' and B' is an LOJ block. Then a subquery SQ of color C that contains all the variable on the outer join side of the LOJ is searched for[6]. If SQ is found in a block B'' it is merged with the LOJ and the subquery associated with its input block. Block B' is removed from AN. If there is a block that has an input p-link from B'' and a color UNKNOWN, it must be revisited to see if it matches Case 1[7].

Case 4: B is an LOJ block with no incoming p-link. This is similar to Case 3.

Step 3: clean up abstract node, consisting of removing subqueries from their associated blocks and removing empty blocks.

END – ABSTRACT NODE MAXIMAL SUBQUERY

We now introduce the last method for a block.

BLOCK MAXIMAL SUBQUERY

Input: a block B with colored variables in variable nodes and propagated variables

Output: a set of source subqueries other than table scans, and a possibly new block B' where subqueries have been removed from B.

Method: traverse in a bottom-up manner the variable dependency graph of the block. At each node of this graph, if (i) its input variables have a same color C distinct from NONE [8], (ii) the node expression is compatible with the capabilities of C, and (iii) it is *profitable* to push the node expression into C, then the node is marked with C. Otherwise, variable equivalences are used to find a set of input variables that would match condition (i). At the end all nodes of a same color C form a subquery (containing a sub-block) for source C and all subqueries other than single table scans are returned.

END – BLOCK MAXIMAL SUBQUERY

The profitability condition of item (iii) above is a heuristics that aims to control that if two subqueries SQ1 and SQ2 of color C can be composed via a node into a subquery SQ of color C, then $dps(SQ) \leq dps(SQ1) + dps(SQ2)$.

The complexity of the search of the maximal subqueries is linear in the number of nodes of a CQP, the number of blocks in an abstract node and the number of nodes in the variable dependency graph of a block.

EXAMPLE 4: We illustrate the application of the method on the example of Figure 1. We apply ABSTRACT NODE MAXIMAL SUBQUERY to abstract node AN = {B1, B2, B3, B4}.

On step1: B3 is the only leaf node and we search for its maximal subqueries using BLOCK MAXIMAL SUBQUERY. We get SQ0 and SQ1 as table scans of T4 and T3 respectively, and [SQ2]S2 = {T3 FOJ T6}. Then B3 takes color NONE.

On step 2: Case 2 applies to B1 and the search for maximal subqueries returns [SQ3]S1 = {F(T1 C T2))} and B1 is marked to UNKOWN. Then Case 4 applies to B4, SQ0 is merged with the LOJ to create [SQ4]S3 = {T4 LOJ T5} which is part of block B3, and B4 is removed. Since B3 → B1 and B1 is UNKNOWN it is examined but remains unchanged. Then Case 4 applies to B2, SQ1 is merged with the LOJ to create [SQ5]S2 = {T3 LOJ T7} which is part of B3, and B2 is removed. Again, since B3 → B1 and B1 is UNKNOWN it is examined and is moved to NONE since it has all its variables.

On step 3: B3 is transformed into {T3 FOJ T4}, B1 is transformed into {E(T2 LOJ T4)}.

So ABSTRACT NODE MAXIMAL SUBQUERY returns AN' = {B1, B3} with four subqueries. Finally, CQP MAXIMAL SUBQUERY resumes by creating three source nodes: SN1 is the result of the merge of T2, T1, and [SQ3]S1; SN2 is the result of the merge of T3, T6, [SQ2]S2, and [SQ5]S2; and SN3 is the result of the merge of T4, T5, and [SQ4]S3. This is illustrated on Figure 2. ¤

[6] If it exists, the subquery SQ if it exists is unique

[7] Because propagation variables have been updated through the merge of the LOJ with SQ.

[8] Getting the color of a variable may recursively trigger the evaluation of the cases of method CQP MAXIMAL SUBQUERY

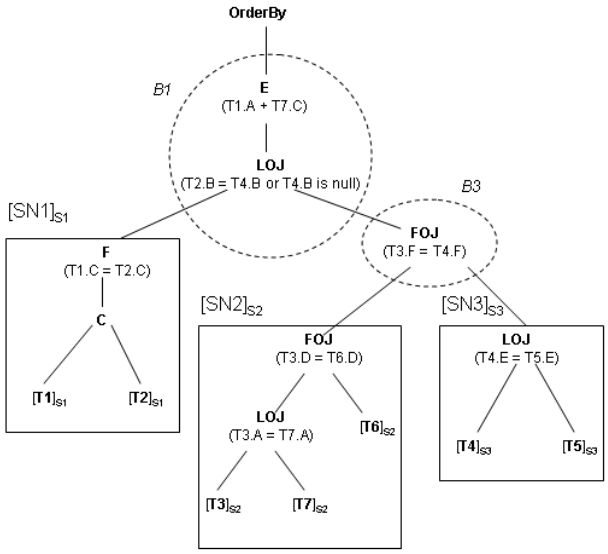

Figure 2: Result of finding maximal subqueries in a CQP

5.2 Semi-Join Reduction in a CQP

5.2.1 Semi-Join Reduction and Bind Joins

Semi-join reduction is a well-known method to execute joins in a distributed system [1].

Suppose that you want to join two subqueries SQ1 and SQ2 from two distinct sources with a join predicate SQ1.A=SQ2.A. The semi-join reduction uses SQ1.A to generate a new subquery:

SQ2': SQ1.A SemiJoin (A=A) SQ2

Then SQ1 will be joined with SQ'2 to produce the final result.

Bind join operations also discussed in [3], [7] have been used as a means to implement semi-join reduction (e.g., in [6]). Reusing the previous example, a possible Bind Join implementation of SQ2' is to:

- collect distinct join attribute values from SQ1 (A values) into a cache node
- create a parameterized query SQ2$ which adds a filter A = $A
- generate as many execution of SQ2$ as there are A values in the cache by substituting the parameter $A with a value from the cache

The principle of semi-join reduction is illustrated on Figure 3, where R denotes the left operand of the bind join operator (noted BJ). We say that the result of SQ1.A is a *semi-join reducer* for subquery SQ2.

The effectiveness of this semi-join reduction depends on the number of distinct A values that will be returned by SQ1 and the number of records returned by each SQ2$ query as compared to the number of records returned by SQ2 (i.e., the cardinality of the result of SQ2). Thus, the decision to use a bind join depends on statistics on distinct values, and the ability to estimate the fan-out factor between SQ1.A and SQ2.A.

We have implemented multiple versions of a Bind Join operator to handle multiple cases of semi-join reductions. In particular, *Hybrid Bind Join* is an implementation that adapts its behavior (hash join or bind join) depending on the size of the distinct values of the cache node. Their discussion is however out of the scope of this paper.

5.2.2 Finding Semi-Join Reducers

Given a CQP, the problem is to find all the bind joins that can possibly reduce the data processing set (dps) of all large subqueries so that the total dps of the CQP remains small.

We first introduce the notion of variable reducer. Let V1 and V2 be two variables of a CQP. Then V1 is said to be a *reducer* for V2 if there exists a filter with a disjunctive term of the form V1 = V2 or there exists an LOJ whose condition has a disjunctive term of the form V1 = V2 and V2 is a variable on the non outer side of the LOJ. The *Reducer Set* for a variable V1 contains all the variables for which V1 can be reducer. If V2 is in this set, we say that V2 is reducible by V1.

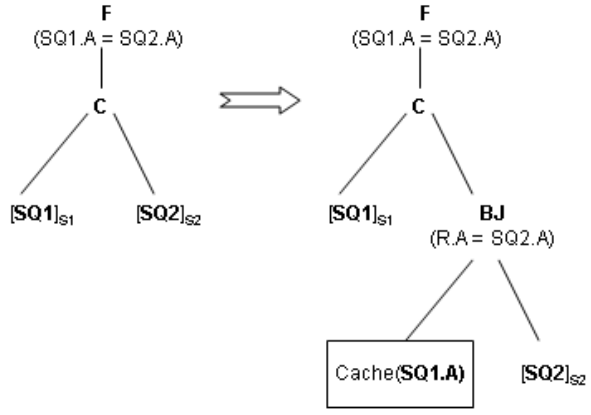

Figure 3: Semi-join reduction using a Bind Join

On the example of Figure 3, variable SQ1.A is a reducer for variable SQ2.A and conversely. Thus, Reducer Set for variable SQ1.A contains SQ2.A.

BUILD VARIABLE REDUCERS

Input: a CQP with a Variable Set containing the output variables of the CQP

Output: a Variable Set containing all the used variables of CQP, and for each variable a Reducer Set that contains its variable reducers.

Method: perform a top down depth first traversal of the CQP as follows. Initially, the Variable Set contains all output variables of the CQP and each variable has an empty Reducer Set.

Case 1: If N is a union operator and V' a variable in Variable Set, then variable input V of N that has been renamed as V' is added to the Reducer Set and inherits the Reducer Set of V'.

Case 2: If N is an aggregate and V is a group by variable of the Variable Set, then the input variable of N corresponding to V inherits the Reducer Set of V.

Case 3: if an abstract node AN is encountered

- If B is a block expression and V is a variable in the Variable Set output of B, each variable V' that is a reducer for V is added to the Reducer Set of V together with its reducers, V' is added to the Variable Set (if it does not exist) and the reducers of V are added to its Reducer Set
- If B is an LOJ block expression and V is a variable in B then V is added to the Variable Set (if it does not exist) and each variable V' that is a reducer for V is added to its Reducer Set together with the reducers of V'.
- If B is a block expression and V is a variable of B that is not in the Variable Set output of B, then V is added to the Variable Set (if it does not exist) and each variable V' that is a reducer for V is added to its Reducer Set together with the reducers of V'.

Case 4: if a source node is encountered, nothing happens.

Case 5: if an FOJnull node is encountered and x is an output variable of the node then x is removed from all existing Reducer Sets.

END – BUILD VARIABLE REDUCERS

We assume that for each node of a CQP, we know (estimate) whether the number of records produced by the node is either "small", "big" or "unknown"[9].

We now introduce a few additional notions. We associate with each node N of a CQP an *annotation* composed of two sets: a *Small Variable Set* (SVS) and a *Big Variable Set* (BVS). The set SVS contains vectors of the form S[x1, …, xk] such that (i) S is an input node of N, and x1, …, xk are output variables of S, and (ii) the number of records produced by N is small. Similarly we define BVS as the set of vectors for input nodes of N that are expected to be big.

We now define a *reduction path* associated with an abstract node as a path of nodes in a CQP whose meaning is that if N precedes N' in the path then N is a reducer for N'. Reduction paths are computed in the main method below and are used to generate the cache nodes of Bind Joins.

We can now present the general method to find semi-join reducers in a CQP. For the sake of clarity, this is a simplified version of the method actually implemented. Refinements do not however change the worst case polynomial complexity of the presented method which is in $o(N^p)$ where N is the number of source nodes and p is the height of the CQP.

BUILD SEMI-JOIN REDUCTIONS

Input: a CQP with its Variable Set and Reducer Set for each variable

Output: a CQP in which all semi-join reducers have been detected and integrated in abstract nodes.

Method: use the following steps:

Step 1: Build the annotations SVS and BVS for every source node of the CQP, and select a source node, InitNode, that has a small dps and whose output variables have at least a non empty Reducer Set[10].

Step 2: perform a bottom-up left to right traversal of the CQP starting from InitNode until no new node can be visited, as follows:

Case 1: if a singleton node N is encountered:

- if N is in {A, U, M, O}, the annotation and the vector for node N are computed in a straight forward manner and classified.
- If N is an FOJnull node then stop traversal and select another InitNode (if any) to perform.

Case 2: if an abstract node AN is encountered:

1. Build (or update) the annotations for AN

2. For each new vector S[x1, …, xk] of SVS(AN), find the vectors of BVS(AN) that are reducible from S[x1, …, xk], henceforth called "big reducible" – see FIND BIG REDUCIBLE

3. For each big reducible vector N' found that is a source node and can be reduced by a vector N to become small, then move N' from BVS(AN) to SVS(AN), add (N, N') to the reduction path of AN ending at N (if any), and build Bind Join for N' using reduction path from N to N'.

4. If a new vector has been added to SVS(AN) then repeat until no new vector is found in SVS(AN):

- find big reducible of each new vector of SVS(AN),
- apply step 3 of Case 2.

5. For each big reducible N' is not a source node and for each variable v of N' that is reduced by a variable of a vector N,

- find the source nodes SN such that SN is big and v is an output variable of SN modulo variable renaming

  if a (single) SN node is found without renaming

  - if the vector of SN is classified as small after reduction then update the annotation of SN and apply Step 2 of BUILD SEMI-JOIN REDUCTIONS from SN as an InitNode until node AN is encountered else return.

  if multiple SN nodes are found modulo renaming

  - Perform a bottom up traversal of the CQP starting from all found SN nodes similarly to Step 2 of BUILD SEMI-JOIN REDUCTIONS except that a union node blocks the bottom up traversal until all its child node's vectors have been updated with respect to the SN nodes found.
- Update the annotation of AN

[9] The notions of "small" or "big" are parameters of the system managed by thresholds. "unknown" is a value between the two thresholds.

[10] Select first the smallest dps and the largest number of reducers

- Add (N, SN) to the reduction path of AN ending at N, and build bind join for SN using reduction path from N to SN.

6. If a new vector has been added to SVS(AN) then:

- find big reducible of the new vector of SVS(AN),
- apply steps 3 to 6 of Case 2.

7. Compute the vector for AN and classify it as big or small

END – BUILD SEMI-JOIN REDUCTIONS

We finally give two auxiliary methods used above.

FIND BIG REDUCIBLE

Input: a vector S[x1, …, xk], and a set BVS(N)

Output: all vectors of BVS that are reducible from S[x1, …, xk], represented in a (big reducible) graph where nodes are vectors and there is an arc from N to N' if N is a reducer for N'

Method: for each variable xi of S[x1, …, xk] search if there exists a variable y in the Reducer Set of xi such that y belongs to a vector N'[…, y, …] of BVS. If found then add vector N' to the output graph and create an arc from S to N'.

BUILD ANNOTATION SOURCE NODE

Input: a source node SN associated with a subquery SQ

Output: annotations SVS and BVS for SN

Method: We use heuristics and statistics to identify if SQ is expected to return a small or large data processing set. If SQ is expected to be small then all its output variables are in a vector SN of SVS, otherwise they are in BVS.

END – BUILD ANNOTATION SOURCE NODE

BUILD BIND JOIN

Input: reduction path from N to N', source node N'

Output: an updated source node of N' which contains a bind join between a cache node and N'

Method: build the maximal subquery covering the nodes of the input reduction path (this is done using the variable dependency graph) and take only the equality predicates that involve variables of N'. This subquery is then projected on the columns that are used in equality predicates with N' with a DISTINCT clause. It is used to compute the cache node associated with N'. Finally the source node N' is rewritten into a bind join between SQ and the cache node.

END – BUILD BIND JOIN

EXAMPLE 5: Consider the CQP of Figure 4. All source nodes have pairwise distinct data sources. Abstract nodes are represented in dotted lines. We assume that S1, R1, and R4 are estimated to be small source nodes, and all the others are expected to be big. We indicate output variables on top of each node. We sketch the application of BUILD SEMI-JOIN REDUCTIONS on the CQP.

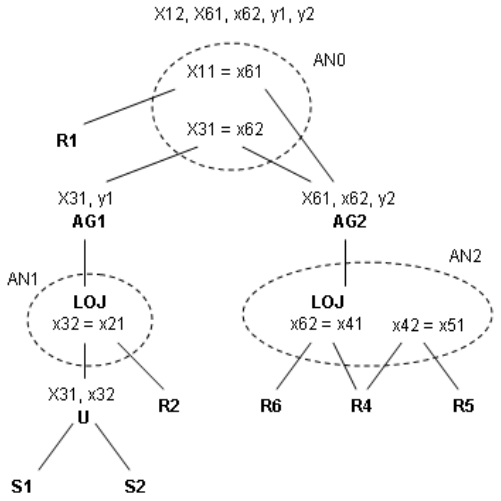

Figure 4: Example of a CQP

Step 1 (R4): S1, R1, and R4 have an empty BVS and their SVS is as follows: SVS(S1) = {S1[x'31, x'32]}, SVS(R1) = {R1[x11, x12]}, SVS(R4) = {R4[x41, x42]}. Suppose InitNode is R4.

Step2 (R4): AN2 is encountered. SVS(AN2) = {R4[x41, x42]}, and BVS(AN2) = {R5[x51], R6[x61, x62]}. R5[x51] is the single vector of BVS that is reducible from R4[x41, x42] because x51 belongs to the Reducer Set of x42. Case 2.3 executes and assuming that the result of the join between R4 and R5 is small then R5 is moved from BVS to SVS, a reduction path (R4, R5) is created and a Bind Join is created in the source node R5. Case 2.5 executes but does not yield anything. Finally, vector for AN2 is AN2[x41, x42, x51, x61, x62] and it is expected to be big

Step 2 (R4): AG2 is encountered. Its vector is AG2[x61, x62, y2] and big.

Step 2 (R4): AN0 is encountered. SVS(AN0) = {R1[x11, x12]}, and BVS(AN0) = {AG1[x31, y1], AG2[x61, x62, y2]}. By Case 2.2, AG2 is the only vector of BVS that is reducible from R1. By Case 2.5, v = x61 is the only variable of AG2 that is reduced by variable x11 of R1. The big source node with output variable x61 is R6 and assuming that R6 becomes small by reduction x11 = x61, then we apply BUILD SEMI-JOIN REDUCTIONS from R6 up to AN0.

Step 2 (R6): AN2 is encountered. SVS(AN2) = {R4[x41, x42], R5[x51], R6[x61, x62]}, and BVS(AN2) = {}. Assuming that the result of the LOJ x62 = x41 is small then vector AN2[x41, x42, x51, x61, x62] is small.

Step 2 (R6): AG2 is encountered. Its vector is AG2[x61, x62, y2] and small. And this finishes the traversal from R2.

Step 2 (R4): Annotation is updated: SVS(AN0) = {R1[x11, x12], AG2[x61, x62, y2]}, and BVS(AN0) = {AG1[x31, y1]}. (R1, R6) is added to the reduction path from R1 and a Bind Join is created in the source node for R6. Case 2.6 executes and AG1 is found as a big reducible from the new addition AG2 in SVS(AN0) via variable x62. Since AG1 is not a source node, Case 2.5 applies. We find two sources nodes S1 and S2 which, after renaming via

the union, have x31 as output variable but only S2 is big. So we restart a variant of BUILD SEMI-JOIN REDUCTIONS from S2.

Step 2 (S2): Union node is encountered, it's a blocker. Since all child nodes vectors have been updated, vector U[x31, x32] is computed and is small.

Step 2 (S2): AN1 is encountered. SVS(AN1) = {U[x31, x32]} and BVS(AN0) = {R2[x21, x22]}. Case 2.2 applies and R2 is reducible from U via x32. Since it is a source node, assuming that R2 becomes small after the reduction by U, then R2[x21, x22] is moved to SVS, a reduction path (U, R2) is added in AN1 and a Bind Join is created in the source node for R2.

Step 2 (S2): AG1 is encountered. Its vector is small. This finishes the traversal from S2.

Step 2 (R4): Annotation is updated: SVS(AN0) = {R1[x11, x12], AG2[x61, x62, y2], AG1[x31, y1]}, and BVS(AN0) = { }. A reduction path (AG2, S2) is added in AN0 and a Bind Join is created in the source node for S2. Case 2.6 does not yield results.

All nodes of the CQP have been visited, so BUILD SEMI-JOIN REDUCTIONS terminates. In summary, all data sources have been reduced by semi-join and the following four bind joins have been created:

(AG2 BJ S2); (U BJ R2); (R1 BJ R6); (R4 BJ R5) ¤

We now use the previous example to make a few observations and point out a few improvements of the method. First, suppose that there exists a new source node R7 that is big and is an input of AN2 with a predicate x52 = x71. Then the reduction path of AN2 (R4, R5) will be extended into (R4, R5, R7) and would be used to compute a Bind Join on R7. Now suppose that R7 is small, then there would be another reduction path (R7, R5) in addition to (R4, R5) and these two paths could be used to build the Bind Join for R5 by building a cache node that contains the Cartesian product of the distinct values of x41 and x71. Finally, assume that the result of the reduction of S2 is uncertain (captured by a value "unknown" on the size of S2 after reduction), if R2 is known to be big it may be useful to create an Hybrid Bind Join for R2.

6. CONCLUSION

We presented the characteristics of operational BI reporting applications and showed that they need to issue queries over virtual target schemas managed by an EII system that federates distributed and heterogeneous data sources. We explained how mega queries over these data sources occur due to the complexity of source to target mappings, thereby raising a significant challenge to EII query optimizers.

We then presented the main decisions taken in the design of the query optimizer of the SAP Business Objects Data Federator system in order to deal with mega queries. We first introduced a new data structure for query plans, called Compact Query Plan (CQP), and then presented two optimization techniques for finding maximal source subqueries and reducing the result set of source queries through Bind Join operations. These two techniques can be run respectively in linear and polynomial time over CQP and consume little memory. They are effective with respect to our goal of minimizing the data processing set of a query, and enable scalable optimization of mega queries.

Several other important aspects of query optimization for mega queries have not been covered in this paper and are just outlined here. First is the use of semi-join reduction to enable dynamic optimization in presence of fragmented data in the sources (expressed via union operation in a CQP). Second is the optimization of the statements generated by a wrapper to a data source taking into account the capabilities of the underlying query engine of the data source (in particular mitigation of complex source queries). Third is the estimation and computation of statistics.

7. ACKNOWLEDGMENTS

The authors want to thank the members of the Data Federator engineering team for their contribution to the overall system and more specifically Raja Agrawal, Benoit Chauveau, Cristian Saita, Fei Sha, Mokrane Amzal, Florin Dragan, Aurelian Lavric, Jean-Pierre Matsumoto, Ivan Mrak, and Mohamed Samy for their great design and implementation effort on Data Federator Query Server. We finally want to thank Dennis Shasha for his regular technical contribution to the team.

8. REFERENCES

[1] Bernstein, P., Chiu, D. Using Semi-Joins to Solve Relational Queries. Journal of the ACM, 28(1):2-40, 1981

[2] Bhargava, G., Goel, P., Bala, I. Efficient Processing of Outer Joins and Aggregate Functions. In Proc. of Int. Conf on Data Engineering (ICDE), 1996.

[3] Florescu, D., Levy, A., Manolescu, I., Suciu, D. Query Optimization in the Presence of Limited Access Patterns. In Proc. of ACM SIGMOD Conf. on Management of Data, 1999

[4] Galindo-Legaria, C., Rosenthal, A. Outerjoin Simplification and Reordering for Query Optimization. In ACM Transactions On Database Systems, 22(1), 1997.

[5] Halevy, A et al. Enterprise Information Integration : Successes, Challenges, and Controversies. In Proc. of ACM SIGMOD Int. Conf. on Management of Data, June 2005.

[6] Manolescu I., Bouganim L., Fabret, F., Simon, E.. Efficient Querying of Distributed Resources in Mediators Systems, Proc of Int Conf CoopIS, Irvine, CA, Oct 2002.

[7] Rajaraman, A., Sagiv, Y., Ullmann, J. Answering Queries using Templates with Binding Patterns. In Proc. of the Symposium on Principles of Database Systems (PODS), San Jose, CA, 1995.

[8] Rao, J., et al. Using EELs, a Practical Approach to Outerjoin and Antijoin Reordering. In Proc. of Int. Conf on Data Engineering (ICDE), 2001.

[9] Rao, J., Pirahesh, A., Zuzarte, C. Canonical Abstraction for Outerjoin Optimization. In Proc. of ACM SIGMOD Int. Conf. on Management of Data, 2004.