Bridging the Gap between OLAP and SQL

Jens-Peter Dittrich[1,*] Donald Kossmann[1,2] Alexander Kreutz[2]

[1]ETH Zurich
Switzerland
www.dbis.ethz.ch

[2]i-TV-T AG
Germany
www.i-tv-t.de

Abstract

In the last ten years, database vendors have invested heavily in order to extend their products with new features for decision support. Examples of functionality that has been added are top N [2], ranking [13, 7], spreadsheet computations [19], grouping sets [14], data cube [9], and moving sums [15] in order to name just a few. Unfortunately, many modern OLAP systems do not use that functionality or replicate a great deal of it in addition to other database-related functionality. In fact, the gap between the functionality provided by an OLAP system and the functionality used from the underlying database systems has widened in the past, rather than narrowed. The reasons for this trend are that SQL as a data definition and query language, the relational model, and the client/server architecture of the current generation of database products have fundamental shortcomings for OLAP. This paper lists these deficiencies and presents the BTell OLAP engine as an example on how to bridge these shortcomings. In addition, we discuss how to extend current DBMS to better support OLAP in the future.

1 Introduction

The key observation that motivates this work is that modern industrial strength OLAP systems implement a great deal of database functionality which would ideally be provided by the underlying database product. A typical and prominent example is SAP's Business Information Warehouse product (BW). Essentially, BW implements a full-fledged query processor on top of the SQL query processor provided by the underlying DBMS. SAP BW is just one example: all OLAP systems we are aware of follow the same approach, in particular, our own product BTell.

It is unfortunate for both sides that OLAP systems make so little use of the functionality of a DBMS, even more so as DBMS vendors have made significant investments in the past to improve OLAP capabilities of their systems [9, 14, 19, 5, 6, 17]. There are historic reasons for this situation [4] because certain developments in OLAP systems precede the latest amendments to DBMSes. There are also technical reasons, due to missing functionality in state-of-the-art DBMS products. In addition, there are also economic reasons because OLAP vendors do not want to become dependent on non-standard functionality provided by certain DBMS vendors.

1.1 Contributions

The purpose of this paper is to explore the missing functionality and show how it can be implemented, using as an example the reporting component of i-TV-T's BTell product. In summary, this paper makes the following contributions:

1. **The Gap:** We list the shortcomings of current DBMS for building OLAP engines and reporting front-ends.
2. **Bridging the Gap:** We present i-TV-T's OLAP and reporting engine as an example on how to bridge these shortcomings.
3. **Closing the Gap:** We present a wish-list on how current DBMS technology should be extended to better support OLAP and reporting front-ends in the future.

Proceedings of the 31st VLDB Conference, Trondheim, Norway, 2005

*Former affiliation, 2003–2004: SAP AG, BW OLAP technology

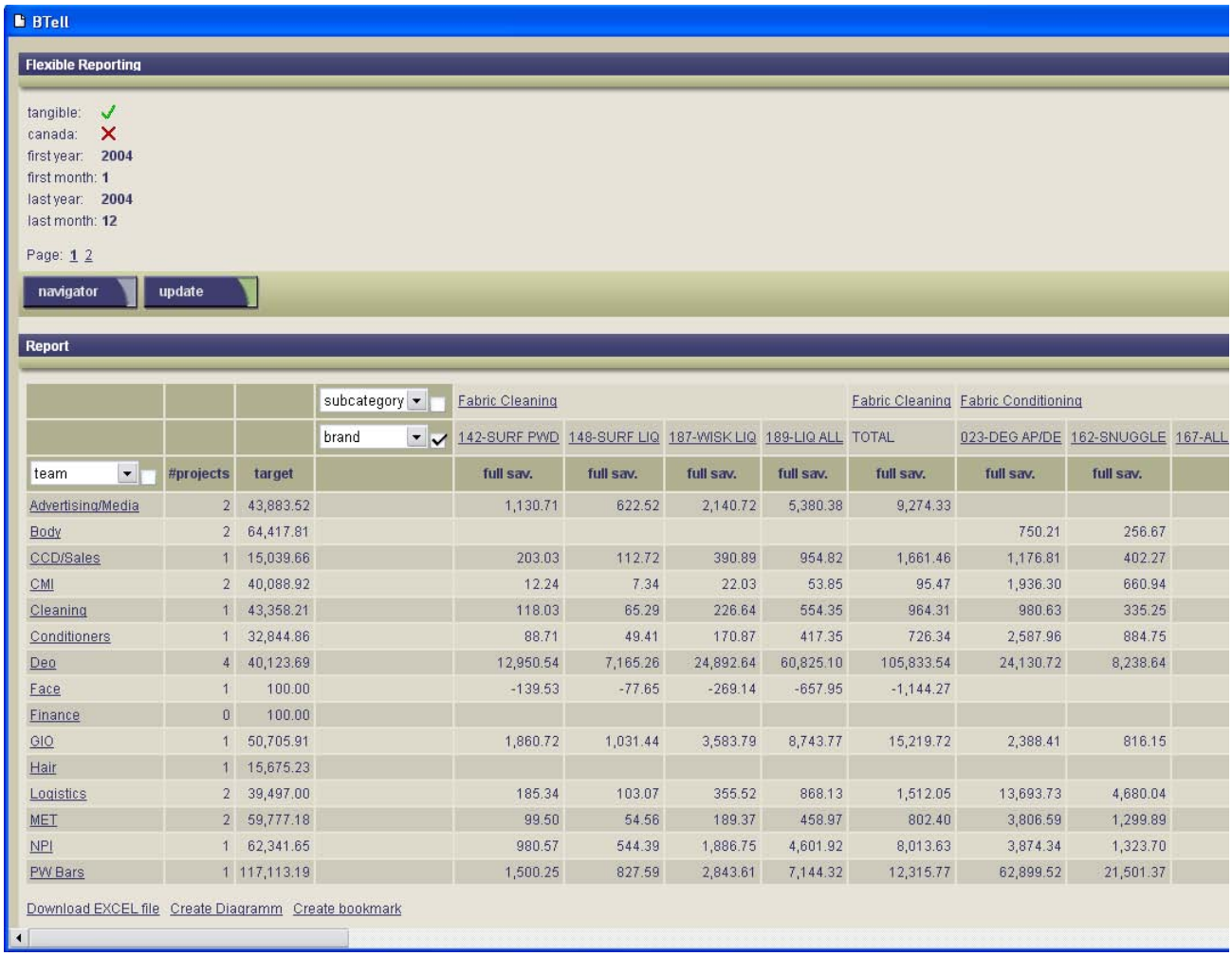


Figure 1: BTell reporting front-end (HTML)

Based on our work, we hope to revive discussions on the suitability of SQL for modern OLAP systems.

This paper is structured as follows: the following section presents the requirements of a modern OLAP system using BTell as an example. After that, Section 3 identifies the problems encountered when building a OLAP and reporting engine on top of current DBMS technology. Section 4 presents how these problems are solved in i-TV-T's BTell product. Finally, Section 5 presents a wish-list on how current DBMS should be extended to better support OLAP.

2 Features of Modern OLAP Systems

As an example for a modern OLAP system, we use the BTell product of i-TV-T AG. BTell is a platform for the development of Web-based information systems. It has been used, among others, for the development of e-Procurement applications (e.g., forecasting, standard cost analysis, factory service agreements, electronic tenders and auctions) and massive multiple-player games (e.g. stock market simulations, business development games). As of 2004, more than 100,000 users in Europe have worked on various applications built on the BTell platform. Currently BTell is used to build a large e-Procurement tool for Unilever in USA, Canada, and Puerto Rico. The applications typically implement a large number of business processes and very complex and flexible reporting. The users range from power users that use the software everyday to users that sporadically use the software, e.g., to download a pre-canned report.

In this work, we focus on the reporting component of BTell which is used to give users a live view on their business data. Figure 1 shows an example report generated by BTell for a 'savings project' application (all numbers are fake). This application manages information of projects that help to reduce the costs of an enterprise. Each project is carried out by a team and reduces costs for products of a particular brand, for a particular factory, in a particular country or business unit, thereby making use of certain strategies (e.g., outsourcing).

The report of Figure 1 shows for each team, its tar-

get savings, the number of projects it is involved in and the actual savings by brand and subcategory. This is a typical report that a user of that application might generate. It shows some features that a modern OLAP system must provide:

a.) **Multi-dimensional Pivot Tables:** In Figure 1, 'sub-category' and 'brand' are pivoted; that is, each brand ('142-SURF PWD', '148-SURF LIQ', etc.) is given its own column, subcategories ('Fabric Cleaning', 'Fabric Conditioning') are represented by a set of columns (one for each brand).

b.) **Moving Sums:** For each subcategory, the total of all savings of all brands in that subcategory is shown. These totals can also be pivoted.

c.) **Split Results:** Depending on user settings, reports are divided into several pages so that the user is not flooded with too much information. In Figure 1, the report is divided into two pages and only the first page is displayed. Users can navigate to the second page by clicking on 'Page 2' in the top part of the page.

Obviously, BTell has a number of features which are not shown in Figure 1, but which are also crucial for the success of a modern OLAP system:

a.) **Interactive Controls:** With simple clicks, more moving sums can be generated, additional metrics can be displayed, and dimensions can be added or removed. Furthermore, pivoting and un-pivoting as well as drill-down and roll-up are controlled by simple GUI features. For example, clicking on 'Body' in the 'team' column will allow the user to get the project information for each member of the team.

b.) **Selection lists:** It is possible to specify selections by the use of condition boxes. For instance, it is possible to generate a report that includes all brands except the brand '148-SURF LIQ'.

c.) **Layout:** Specific color encodings (e.g. traffic lights) can be used in all reports. Furthermore, reporting orders can be redefined (e.g., group the teams according to certain criteria rather than listing them in alphabetical order).

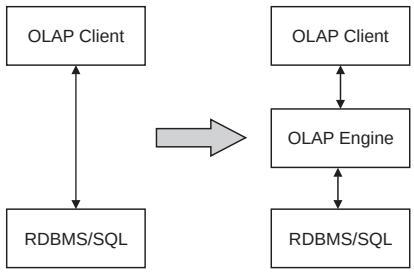
d.) **Downloads, Graphics:** A report can be downloaded to Excel. Furthermore, bar charts, pie charts, speedometers, etc. can be generated.

e.) **Pre-canned reports:** The reports can be stored as bookmarks and then be re-evaluated with a simple click. Furthermore, bookmarks can be sent to other users (e.g. managers) by email so that these users can trace the latest results.

While there has been significant progress on DBMS products, e.g., on Pivot Tables [3] and integration of spreadsheet functionality [19], this progress is not enough in order to implement all these OLAP features directly using a DBMS. There are fundamental shortcomings which will be described in the next section.

3 The Gap: Why SQL is not Enough

Most of today's OLAP platforms rely on a relational database (ROLAP) which is used to store a historical snapshot of integrated data from several underlying OLTP systems. The snapshot is either stored in specialized schemas like the *Star or Snowflake Schema*; or in flat views like *Operational Data Stores* (ODS). The functionality of the RDBMS is extended by each OLAP vendor (like SAP or i-TV-T) through a proprietary OLAP engine built on top of the RDBMS as displayed in the following Figure:



This architecture is used to perform a two-step (filter/refine) data processing strategy:

1. **Filter:** The RDBMS retrieves a superset of the data that is actually needed. The RDBMS is only used to perform heavy data processing tasks like pre-aggregation and joins.

2. **Refine:** The OLAP engine uses the superset to compute the exact result to each query.

There are several reasons why vendors choose a two-step architecture:

1. Though SQL has been extended with a variety of important new OLAP operators, e.g. the Cube [9], these operators are still not provided with each RDBMS. Therefore, OLAP vendors tend to support only the minimal set of SQL that is supported by all RDBMS vendors.

2. Even systems that implement the latest SQL standard lack important OLAP features. As a consequence, system architects use only the common set of functionality that is provided by all RDBMS vendors. Everything else will be implemented inside the OLAP engine, even those tasks that could be performed by certain RDBMS products.

Profits

State	Customer	Product	Profit
S1	C1	P1	1.0
S1	C1	P2	1.0
S1	C1	NULL	2.0
S1	C2	P1	1.0
S1	C2	P2	1.0
S1	C2	NULL	2.0
S1	NULL	NULL	4.0
S2	C1	P1	1.0
S2	C1	P2	1.0
S2	C1	NULL	2.0
S2	C2	P1	1.0
S2	C2	P2	1.0
S2	C2	NULL	2.0
S1	NULL	NULL	4.0
NULL	NULL	NULL	8.0

a) The result of a ROLLUP operation

Profits

State	Customer	Product	Profit
S1	C1	P1	1.0
S1	C1	P2	1.0
S1	C1	∑	2.0
S1	C2	P1	1.0
S1	C2	P2	1.0
S1	C2	∑	2.0
S1	∑∑		4.0
S2	C1	P1	1.0
S2	C1	P2	1.0
S2	C1	∑	2.0
S2	C2	P1	1.0
S2	C2	P2	1.0
S2	C2	∑	2.0
S2	∑∑		4.0
∑∑∑			8.0

b) Interpreting NULL-values as multi columns

Profits

State	Customer	Product	Profit
S1	C1	P1	1.0
		P2	1.0
		∑	2.0
	C2	P1	1.0
		P2	1.0
		∑	2.0
	∑∑		4.0
S2	C1	P1	1.0
		P2	1.0
		∑	2.0
	C2	P1	1.0
		P2	1.0
		∑	2.0
	∑∑		4.0
∑∑∑			8.0

c) Interpreting adjacent similar values as multi rows

Figure 2: The result of a ROLLUP and its 'interpretations'

In summary, commercial OLAP engines tend to re-implement considerable database functionality. They perform database-like tasks like pivot computation, post-aggregation, hierarchy operations, semantic correctness checks, caching, etc. The OLAP engines bridge the gap between the relational world of the RDBMS and the multidimensional analysis required by the user.

The following sections (3.1–3.3) identify three of these gaps. After that, Section 4 present how these gaps are bridged in BTell. Finally, Section 5 presents a wish-list on how to close the gap, i.e., how to extend current RDBMS to better support OLAP in the future.

3.1 Non-Relational Data Model

This section shows that the tabular relational model is not always suitable for OLAP because OLAP systems must present query results as part of a GUI. We argue that a non-relational, cell-oriented representation of data is more appropriate to present query results than the relational model. Furthermore, the relational model is not able to unambiguously represent certain values.

SQL 99 introduced two new operators for OLAP: CUBE and ROLLUP [9]. These operators compute multiple groupings as well as intermediate aggregates and sums. The difference between the two operators is that CUBE creates all existing aggregates whereas ROLLUP creates only the subset of CUBE corresponding to a hierarchy of columns.

For example, if we do a ROLLUP on State, Customer and Product, i.e.,

```
SELECT State, Customer, Product, sum(Profit)
FROM Profits
GROUP BY ROLLUP (State, Customer, Product)
ORDER BY State, Customer, Product;
```

we receive the table displayed in Figure 2a.

Example 1: (Multi Column Results) Figure 2a contains all rows from the base table Profits as well as additional rows containing NULL-values. These NULL-values have to be 'interpreted' as sums, since SQL does not provide a special format for sum. Figure 2b shows, how these NULL-values are interpreted over one or multiple columns, respectively. The problem is that SQL also uses NULL-values for outer joins. In this case, the NULL-value is interpreted as 'value does not exist'. To disambiguate between the two different semantics of the NULL-value, SQL 99 introduced a special column function named GROUPING(). If GROUPING() is called with a NULL-value representing a sum, 1 is returned, 0 if it has different semantics. Since OLAP-queries typically contain outer joins, GROUPING() has to be used with a combination of CASE to ensure correctness. This makes SQL cumbersome and error-prone and simply not expressive enough for OLAP applications.

Example 2: (Multi Row Results) The ROLLUP operation in the previous example used an ORDER BY statement to sort the relation lexicographically on columns State, Customer and Product. Note, that relations are defined as sets, i.e., Profits $\subseteq$ State $\times$ Customer $\times$ Product $\times$ Profit. If we sort a relation into a sequence, it is not a relation anymore. In other words, a relation is *not* a sequence but a set.

Figure 2b shows the sorted output of the ROLLUP-operation. Many key columns contain similar values in consecutive rows, i.e., similar values are repeated for each row. This is another interpretation convention of SQL. It means, that these values represent an entry that spans *multiple rows*, i.e., a *multi row entry*. Figure 2c visualizes this interpretation. Adjacent similar values are merged to form a multi row cell.

These multi row entries are neither supported by SQL nor by the relational model.

Example 3: (Column Orders) Figure 2c shows a drill-down by State, Customer and Product. In other words: Profits are first drilled-down by State, then each value of the State column is drilled-down by Customer. After that, each value of the Customer column is drilled-down by Product.

If the columns were in a different order[1], say Customer, State, Product, we would see a different table. The order of columns implicitly defines a 3-level hierarchy, where State is the root and Product the leaf level. Neither the order of columns nor inter-column hierarchical dependencies are part of the relational model.

Example 4: (Pivot Tables) A pivot table is a 2-dimensional representation that displays values on both the x- and the y-axis. For example, a pivot table with State and Customer on the y-axis and Product on the x-axis looks as follows (another example is given in Figure 1):

Profits		Product		
State	Customer	P1	P2	∑
S1	C1	1.0	1.0	2.0
S1	C2	1.0	1.0	2.0
S1	∑	2.0	2.0	4.0
S2	C1	1.0	1.0	2.0
S2	C2	1.0	1.0	2.0
S2	∑	2.0	2.0	4.0
∑∑		4.0	4.0	8.0

The above issues on multi row results, multi column results and column orders fully apply to pivot tables. However, since pivot tables can be seen as a 2-dimensional extension of a roll-up, things get even more complicated. For example, the pivot table contains three columns for the Profit measure, one for each value appearing in the Product column and a totals column. In addition, the pivot table in the example contains more aggregate values. For instance, the values ((S1—S2),∑,(P1—P2)) are not part of the `ROLLUP` in Figure 2.

Currently, SQL does not support pivot tables. Recently, two new operators `PIVOT` and `UNPIVOT` have been proposed as first-class RDBMS operators [3]. However, the proposal in [3] is not sufficient because columns must be explicitly defined as part of the query, i.e., no dynamic pivot tables are allowed. In addition, only one pivot dimension is possible. Given these shortcomings and since only one DBMS vendor has started to work on this topic, OLAP vendors are forced to implement this important feature in their OLAP engine.

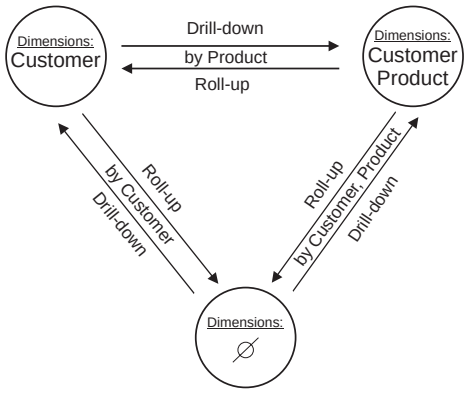
The next section will explore additional deficiencies of SQL that are related to the client-server architecture of multi-tiered OLAP systems.

3.2 Client/Server Architecture

This section shows that the current client/server computing model has problems when used for OLAP.

Example 5: (Navigation) An important paradigm of OLAP is the concept of *navigation*. Typically, a user starts analyzing data by selecting an initial query. This query consists of a set of dimensions on the x- and the y-axis as well as a set of filter conditions. After that, the user modifies the query in an interactive fashion by adding or removing columns (*drill-down* and *roll-up*), adding or removing filter conditions (*slicing*), moving columns from the y- to the x- axis (*dicing*) and so on.

This navigational pattern is best described by a graph representation, where the current selection of dimensions and filter conditions corresponds to a node, i.e., the current state of the OLAP query. The edges represent transitions between different states:



The transition from one state to another is unambiguously defined by the parameters of the transition. It is *not* necessary to resubmit the *entire* query.

In contrast, the query language that is used to declare OLAP queries, SQL, is stateless by definition. At each step, the entire query is resubmitted again and again. No knowledge of previously submitted queries of a user is preserved.

Example 6: (Caching) In a three-tier architecture, each tier (client, application server and DBMS) maintains separate caches. These caches are used to store data received from other tiers or results computed for other queries. All caches and *data stores* outside the DBMS must be kept in sync manually. This again is labor intensive and error-prone.

Figure 3 depicts the three tiers as well as their associated data stores and caches. The DBMS stores the base tables. These tables are joined and aggregated to compute views. Some of them are stored, i.e. *materialized* [10], on the *DBMS tier*. Then, the OLAP engine at the *application tier* stores a subset of these views. It uses them to compute cubes, roll-ups and pivot tables. Some of the results are stored in a separate cache on the application tier. Finally, the *client tier* stores a subset of the results computed by the OLAP engine. The client further processes the data to produce formatted reports in HTML, XML or MS Excel. Some of these reports are also cached on the client tier.

The Figure shows that, from a bird's eye perspective, all three tiers perform the same task:

1. Receive and store some input data.

[1]A table with n key columns has $n!$ different column orders.

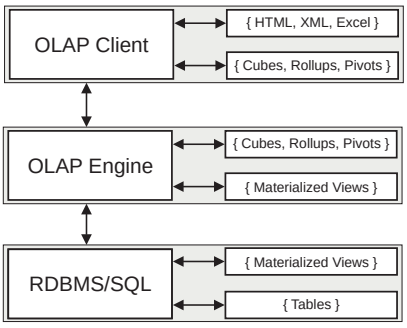


Figure 3: The 3 tiers and their associated caches and data stores

2. Perform algebraic query processing and optimization on the data.

3. Store some output data, send some of it to other tiers.

The punch line is that each tier has specific caching logic and that the DBMS cache which would be useful for all tiers is in the DBMS cage and cannot be used on the different tiers. As a result, standard DBMS logic must be replicated at all tiers.

3.3 Computability of Aggregates

In this section, we show that the `GROUP BY` statement does not always produce the correct result. This problem is called the problem of *summarizability* and was first identified in [18]. [16] presents a detailed overview on the problem and provides three necessary conditions for summarizability. [11, 12] study summarizability for selected classes of heterogeneous schemas.

In order to implement summarizability correctly, the DBMS must be aware of the functional dependencies between columns, even in those cases, where base tables are joined and aggregated to create new views: it is not enough to consider functional dependencies on the base relations only. Since current DBMS optimizers do not support this feature, *summarizability awareness* has to be provided by the OLAP engine. The following examples show how the lack of this feature results in wrong query results.

Example 7: (Unexpected Results) Consider a resource planning application of a freight shipping company. The following Figure shows a table `Trucks`, containing data on trucks and their capacity; and a table `Transports`, containing company names, city and the required capacity.

```
SELECT     Make, sum(capacity)
FROM       (   SELECT *
               FROM Trucks NATURAL JOIN Transports )
GROUP BY   Make;
```

Trucks

Make	Capacity
Ford	10.0
VW	10.0

Transports

Company	City	Capacity
BigComp	NYC	10.0
BigComp	LA	10.0

```
↓  SELECT *
   FROM Trucks NATURAL JOIN Transports;
```

PossibleTransports

Make	Company	City	Capacity
Ford	BigComp	NYC	10.0
Ford	BigComp	LA	10.0
VW	BigComp	NYC	10.0
VW	BigComp	LA	10.0

```
   SELECT Make, sum(Capacity)
↓  FROM PossibleTransports
   GROUP BY Make;
```

Trucks (Aggregate)

Make	Capacity
Ford	**20.0**
VW	**20.0**

Both tables are joined by a natural join[2]. The resulting table `PossibleTransports` contains a list of possible truck-transport pairs.

Now, the user asks for a `GROUP BY` on the Make column of `PossibleTransports` using `sum` as the aggregation function. The resulting table `Trucks (Aggregate)` has the same structure as the source table `Trucks`. However, it contains different, i.e. unexpected, data entries in the Capacity column; for example, the query result indicates that the capacity of a Ford is 20 whereas it really is only 10.

The source of this irritation is that the `Possible-Transports` table, an intermediate query result, is not normalized and, thus, contains data redundancies. If such an intermediate query result is aggregated, data items from the base tables are used multiple times. As a consequence, the measures in the result of the `GROUP BY` query are wrong. Modern OLAP systems can detect such situations and are able to compute the expected result; this, however, comes at the additional price to carry out the aggregation in the OLAP engine, rather than pushing the whole query down to the DBMS.

Example 8: (Sales) OLAP measures are typically related to a unit and only values of the same unit can be aggregated. The awareness for units is another feature that SQL and state-of-the-art DBMS are lacking in order to adequately support OLAP applications. In

[2]The natural join is used for sake of simplicity. In a real application, we would perform a theta join using θ = `Trucks.Capacity` $\geq$ `Transports.Capacity` as the predicate.

the following, we will consider a sales application with the following simple table representing sales profits:

Profits				
State	Customer	Product	Profit	Unit
S1	C1	P1	1.0	MM USD
S1	C1	P2	1.0	MM USD
S1	C2	P1	1.0	MM USD
S1	C2	P2	1.0	MM EUR
S2	C1	P1	1.0	MM EUR
S2	C1	P2	1.0	MM USD
S2	C2	P1	1.0	MM EUR
S2	C2	P2	1.0	MM EUR

The following query should result in an ERROR:

```
SELECT Customer, sum(Profit)
FROM Profits
GROUP BY Customer;
```

However, standard DBMS will execute this query and return the wrong result. (Alternatively, a user-defined aggregation function needs to be executed.) Again, an OLAP system will consider the peculiarities of units and make sure that all aggregates are carried out correctly.

4 BTell: How to Bridge the Gap?

Ideally, today's RDBMS products and SQL should be extended to solve the problems mentioned in the previous section. The relational model should be extended to support order, hierarchies, multi-columns, multi-rows and multi-dimensional concepts like pivot 'tables'. The client/server paradigm should be broadened to provide support for a more open query processing model. Last but not least, SQL should be extended to model functional dependencies and units — not only at table, but also at query, view and result level — in order to guarantee correct results.

Since all this is not likely to happen in the near future, OLAP vendors, like i-TV-T, have invented their own solutions. In the following, we sketch some of BTell's solutions to these problems.

4.1 Operator Model

In this section, we first introduce the multi-dimensional operator model of BTell. After that, we explain how BTell's operators are used to provide efficient caching and pivot computation.

Query processing in BTell is based on an operator model. In contrast to standard relational operators, we distinguish two classes of operators:

1S The first class takes one or more input stream(s) and returns a single output stream. These operators are non-blocking operators, i.e., iterators that comply with the open-next-close interface [8].

3S The second class takes one or more input stream(s) and returns three output streams. These operators are blocking operators.

The streams of the 1S operators contain mixed data that may apply to one of the axis' or both of them. In contrast, the three streams of the 3S operators have the following semantics: the first stream contains data for the x-axis; the second, data for the y-axis; and the third stream data, that applies to both the x-axis and the y-axis.

Example: Recall the pivot table from the previous examples:

Profits		Product		
State	Customer	P1	P2	$\sum$
S1	C1	1.0	1.0	2.0
S1	C2	1.0	1.0	2.0
S1	$\sum$	2.0	2.0	4.0
S2	C1	1.0	1.0	2.0
S2	C2	1.0	1.0	2.0
S2	$\sum$	2.0	2.0	4.0
$\sum\sum$		4.0	4.0	8.0

For this pivot table, BTell generates an x-stream containing tuples (S1,C1,2.0), (S1,C2,2.0), (S1,$\sum$,4.0), etc.; a y-stream containing tuples (P1,4.0), (P2,4.0); and an xy-stream containing tuples (S1,C1,P1,1.0), (S1,C1,P2,1.0), (S1,C2,P1,1.0), (S1,C2,P2,1.0), etc., respectively. This means, BTell splits the pivot into cells that are valid only on the x-, y-, or on both axis. This strongly facilitates pivot computation as will be explained in Sections 4.4.

4.2 Pipelining

Figure 4 shows the pipeline of BTell OLAP. The operators displayed as boxes are 3S operators. The operators displayed as circles are 1S operators.

Query processing is triggered by the clients. The client creates a `ReportOptions` instance, which collects the parameters of the query, like dimensions to display, filter conditions, etc. The `ReportOptions` are sent to BTell's OLAP engine which passes them to the top-level operator of the pipeline, e.g., the Excel Convert operator. For the moment, we will assume that the pipeline does not contain cached data. Therefore, each operator will call its parent operator until the `Fetch` operator is reached. The `Fetch` operator sends SQL-queries to the RDBMS and retrieves the result rows. The result is then split into three streams (x, y and xy) and sent to the next operator. The next 3S operators perform caching, filtering, sorting and grouping. After that, 1S operators perform pivot computation and post-processing on the pivot tables.

4.3 Caching

The main idea of caching in BTell is to mix standard query operators and special OLAP operators with caching operators[3]. Therefore,

[3]The same approach has recently been applied in a different context to better utilize instruction cache performance [20].

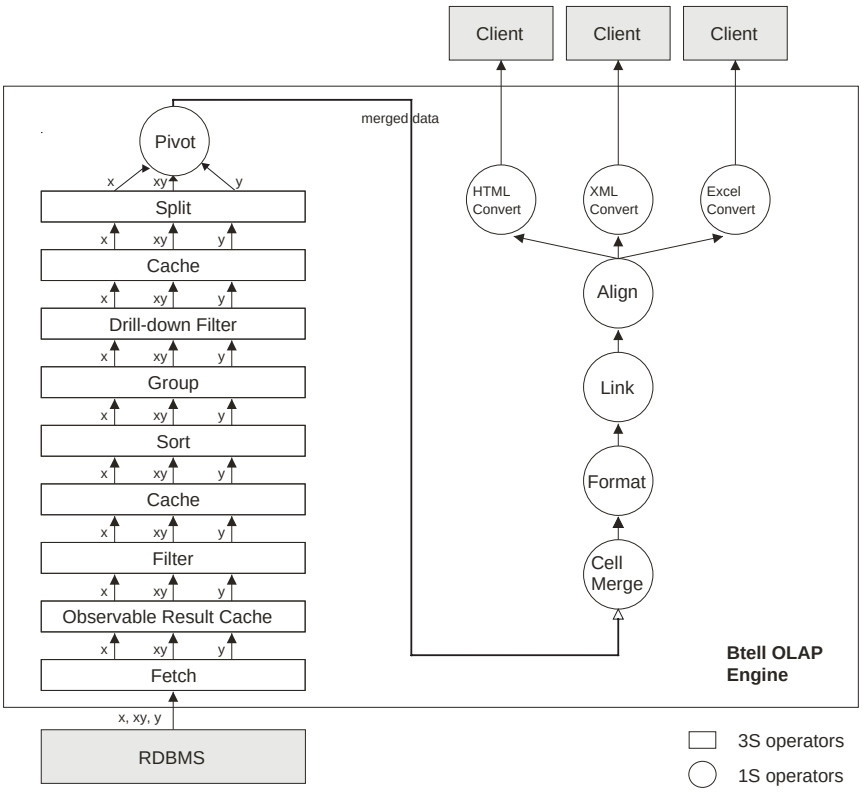


Figure 4: The processing pipeline of BTell's OLAP Engine

all 3S operators support a special operation `must_reevaluate(ReportOptions)`. The semantics of this operation are as follows: each time the report options change `must_reevaluate(ReportOptions)` may be called on an operator to determine whether that operator would now compute a different result for the given options. In the latter case, `true` is returned, `false` otherwise. Like that, `must_reevaluate` weaves caching into the operator model. This greatly facilitates the implementation of the cache update policies.

4.4 Pivot Tables

We will now sketch the algorithm that is used by the 3S operators to compute pivot tables.

4.4.1 Pivot Computation

The following algorithm extends the algorithm proposed in [9] to the 3S operator model. The main idea of the our algorithm is to apply two lexicographical sorts on the data. Assume we have a table with columns $c_1,..,c_k,c_{k+1},..,c_d$, where `d` is the total number of columns. We want to compute a pivot table with columns $c_1,..,c_k$ on the y-axis and columns $c_{k+1},..,c_d$ on the x-axis, respectively. The pivot algorithm works as follows:

1. **Y-Sort**: Sort the data using columns $c_{k+1},..,c_d,c_1,..,c_k$ as the lexicographical compare order.

2. **Y-Group**: Compute moving sums of all rows by simply iterating column-wise through the data. Each group change generates a moving sum.

3. **X-Sort:** Sort the data plus the newly created moving sums using $c_1,..,c_k$, $c_{k+1},..,c_d$ as the lexicographical compare order.

4. **X-Group:** Compute moving sums of all columns by simply iterating row-wise through the data. Each group change generates a moving sum.

Steps 3 and 4 can be swapped with 1 and 2. This will produce the same result.

We will now briefly discuss the pivot algorithm as implemented on top of the 3S operator model. The `Sort` and `Group` operators create three separate output streams of data. The y-stream contains tuples representing the keys displayed on the y-axis as well as the measure `m`; the x- and xy-streams contain keys displayed on the x- and xy-axis, respectively. Thus, y-stream tuples have then format $(<c_1,..,c_k>, m)$; the x- and xy- streams have then format $(<c_{k+1},..,c_d>, m)$ and $(<c_1,..,c_k,c_{k+1},..,c_d>, m)$, respectively. For this reason, the actual `Pivot` operator simply performs a merge of the three streams using columns $c_1,..,c_d$ as the join

keys. The latter join is implemented as a three-way mid[4]-outer sort-merge join.

4.4.2 Post-processing

The `Pivot` operator generates a sorted stream of cells with format (<c_1,..,c_k,c_{k+1},..,c_d>,<cell object>) as its output. After that, the cells are enriched by the 1S operators. For example, the `Format` iterator converts numbers and dates to formatted output strings applying the user's regional settings; the `Align` iterator, aligns data cells to the right or left margin. Finally, three different `Convert` operators convert the cells to either HTML, XML, or MS Excel.

4.5 Computability of Aggregates

In this section, we present BTell's algorithm for determining summarizability of aggregates. To the best of the authors' knowledge, BTell is the only product that performs such kind of summarizability check.

4.5.1 Main Algorithm

The main idea of our algorithm is as follows: First, the data model as well as functional dependencies between columns have to be declared in the data dictionary. Second, at runtime the functional dependencies are exploited to determine whether the current drill-down is valid.

Recall the table from the running example[5]:

Profits			
State	Customer	Product	Profit
S1	C1	P1	42.00
S1	C1	P2	42.42
S1	C2	P1	11.00
S1	C2	P2	5.00
S2	C1	P1	42.00
S2	C1	P2	42.42
S2	C2	P1	11.00
S2	C2	P2	5.00

For this table, a user might declare a list of functional dependencies as follows:

$$\{\text{Customer, Product}\} \rightarrow \text{Profit}$$

This means, Customer and Product determine the measures column Profit. But, the column State neither determines any other column nor does it depend on any other column.

The following table shows the aggregates, i.e. moving sums, that are valid for this example:

Profits				
State	Customer	Product	Profit	is valid?
.	.	.	.	YES
.	.	Σ	.	YES
.	Σ		.	YES
Σ			.	↯ NO ↯

[4] `outer` is applied to the xy-stream only.

[5] We have changed the numbers in the 'Profit' column to avoid the trivial functional dependencies caused by a constant value.

ALGORITHM FunctionalDependencyCheck

Input:
-columns c_0,...,c_d
-measure m
-set of functional dependencies F1: $\{c_i \rightarrow c_j\}$
-set of functional dependencies F2: $\{c_j \rightarrow m\}$

Output:
-interval T of dimensions, where totals are allowed

```
(1)  Using F1 and F2 compute minimal set
           of columns A={c_i} that determines m
(2)  compute closure A+ of A
(3)  start_dim = d + 1
(4)  ForEach i in {d,...,0}:
(5)      If c_i ∈ A+:
(6)          start_dim = i
(7)      Else
(8)          break
(9)      EndIf
(10) EndFor
(11) T = (start_dim, d)
(12) return T
```

Figure 5: `FunctionalDependencyCheck` Algorithm

BTell's `FunctionalDependencyCheck` algorithm is depicted in Figure 5. It is invoked with a list of columns c_i to check, one measure `m` and two sets of functional dependencies: one containing dependencies between columns, the other containing dependencies between columns and the measure. The extension of the algorithm to multiple measures is straightforward and omitted for reasons of readability. The algorithm returns an interval `T` that contains the range of columns where movings sums are valid.

The algorithm works as follows: it starts by computing the minimal set of columns `A` that determine the measure `m` (line 1). After that, the closure A^+ of `A` is computed, i.e., all functional dependencies that are implied by `A` (line 2). The variable `start_dim` is used to store the first column that may be aggregated. It is set to `d+1` (line 3). Then, the algorithm iterates over the columns starting at the rightmost column (lines 3–10). If the current column is determined by the closure A^+ (line 4), the iteration continues and sets `start_dim` to the current column index (line 5). Otherwise, the iteration halts (line 8). The algorithm returns the interval `T=(start_dim,d)` as the result, where `start_dim` refers to the last valid column that was checked in the for-loop (lines 11–12).

For our example, the algorithm would compute `A` as `A = {Customer,Product}`. Then, the cover A^+ would be computed as A^+ `= A`. The iteration starts with column Product. Product is contained in the cover set. Therefore the iteration continues. The next column to check is column Customer. Again, this column is contained in the cover set A^+. The next column State, however, is not contained in the cover set. Therefore,

the iteration halts. `T=(1,2)` is returned as the result. This is the expected result.

4.5.2 Extensions

There are some important extensions that have to be considered when implementing `Functional-DependencyCheck`.

Multiple Minimal Sets There are situations, in which multiple minimal sets exist that determine the measure `m` (compare line 1 in Figure 5, also see Example 7). For these situations, the algorithm must not allow moving sums on the entire range of columns. Therefore, `T = (d+1,d+1)` should be returned.

Pivot Tables For pivot tables, the algorithm is applied separately on both drill-down dimensions (x-axis and y-axis). The result is then combined to determine placement of moving sums. Note, that a moving sum might be valid on both axis, on one axis or none of them.

Units Units are treated separately. There are two cases: if no unit is present, or the input data set is restricted to tuples that all have the same unit, values can safely be aggregated. Otherwise, aggregation is restricted to those columns that have the same unit, or can be converted to a common unit.

5 How to Close the Gap?

The previous sections have identified the gap between OLAP and SQL and showed how this gap is bridged in a commercial product. In this section, we will explore how to close the gap, i.e., how to extend DBMS to better support OLAP and reporting technology in the future. The extensions proposed here are not part of the BTell product; they are, however, currenty discussed as future development directions of our product.

We think there are two paths to follow: the first is to extend SQL with new OLAP features. This will help to close a lot of gaps like summarizabilty, unit handling, pivot computation and so on. On the other hand, it is hard to extend SQL with features to represent non-relational, multidimensional data (compare Section 3.1). Though the latter could be accomplished by, e.g., using nested relations, handling OLAP queries in SQL then would not become much easier.

The second path to follow is to develop a new query language designed for OLAP from the beginning[6]. This new language should be standardized. DBMS vendors should then provide add-on products to their DBMSes that translate SQL to that new language and vice versa. This would, in the long-run, remove the need to implement proprietary OLAP engines.

[6]Microsoft has already developed a proprietary language called MDX (Multi-Dimensional eXpressions). Though MDX helps to fix some of the problems with SQL, many of the gaps presented in this paper are not tackled.

In this section, we will sketch how this new query and data definition language could look like. First of all, recall that only small amounts of data are transfered between the users client and the OLAP engine. The heavy data processing tasks are performed only inside the DBMS or inside the OLAP engine. For this reason we choose XML as the data return format — the overhead introduced by XML will not substantially decrease the performance of our system proposal. Also note, that XML can already be processed by a huge number of reporting tools. To provide efficient query processing on XML data XQuery is currently developed to become the lingua franca of the XML world. Just recently powerful OLAP extensions have been proposed to facilitate analytic queries with XQuery [1].

5.1 Wish List for an Analytical Query Language

We will now sketch how an *Analytic Query Language (AQL)* should be designed to not only bridge but close the gaps described in the previous sections of this work. We assume that XQuery plus the OLAP extensions proposed in [1] will build the foundation for such a language. We do not present a complete specification of these extensions here. We think that this should be accomplished by the research community and the W3C XQuery committee. Our primary goal here is to stimulate discussion on the topic. The following items represent our wish list:

1. AQL should represent all data and metadata available in the DBMS as XML views. Note, that the data has neither to be stored nor processed in XML format inside the DBMS. We only require XML views in order to provide unified access to the data.

   Impact: This allows to perform all data definition tasks using XQuery[7].

2. AQL should be extended to enable abstract data definitions.

   Impact: This allows to model semantic relationships as well as functional dependencies.

3. AQL should provide a facility to place, i.e. drill-down, attributes on 'rows' or 'columns' (just like MDX).

   Example statement:

   ```
   for $f in //profits
   group by $f/state, $f/customer ON ROWS,
   $f/product ON COLUMNS
   return ...
   ```

   Impact: This strongly facilitates the semantics of OLAP queries.

[7]We assume that `update` and `insert` operations will become available in XQuery in the near future.

4. AQL should provide operators to automatically generate multi-dimensional pivot, cube and rollup representations.

   Example statement:

```
for $f in //profits
group by ROLLUP ($f/state, $f/customer) on rows,
ROLLUP ($f/product) on columns
return ...
```

   Impact: This allows to easily compute rollup, cube and pivot representations.

5. AQL should provide multi-dimensional return formats.

   Example statement:

   Lets assume we want to compute the following pivot result:

Profits		Product		
State	Customer	P1	P2	$\sum$
S1	C1	1.0	1.0	2.0
S1	C2	1.0	1.0	2.0
S1	$\sum$	2.0	2.0	4.0
S2	C1	1.0	1.0	2.0
S2	C2	1.0	1.0	2.0
S2	$\sum$	2.0	2.0	4.0
$\sum\sum$		4.0	4.0	8.0

   We propose, that the statement

```
for $f in //profits
group by rollup ($f/state, $f/customer) on rows,
rollup($f/product) on columns
return AS MDVIEW
```

   creates the following result:

```
<profits>
  <rows>
    <S1>
      <C1>  <1/><2/><3/> </C1>
      <C2>  <4/><5/><6/> </C2>
      <sum> <7/><8/><9/> </sum>
    </S1>
    <S2>
      <C1>  <10/><11/><12/> </C1>
      <C2>  <13/><14/><15/> </C2>
      <sum> <16/><17/><18/> </sum>
    </S2>
    <sum> <19/><20/><21/> </sum>
  </rows>
  <columns>
    <P1> <1/><4/><7/><10/><13/><16/><19/> </P1>
    <P2> <2/><5/><8/><11/><14/><17/><20/> </P2>
    <sum> <3/><6/><9/><12/><15/><18/><21/> </sum>
  </columns>
  <data>
    <1> 1.0 </1>
    <2> 1.0 </2>
    <3> 2.0 </3>
    <4> 1.0 </4>
    ...
    <21> 8.0 </21>
  </data>
</profits>
```

   Note, that this format preserves both hierarchies, as well on the x- as well on the y-axis. In addition, no result tuples of the aggregation get repeated.

   Impact: This allows to compute query results that can easily be postprocessed by a client application.

6. AQL should allow to modify existing queries. In addition, stateful queries and sessions should be possible.

   Example statement:

```
DEFINE SESSION $s AS
       for $f in //profits
       group by $f/state on rows,
       $f/product on columns
       return as mdview

$ret = EVAL($s)
```

   Note, that the `DEFINE` command does not compute any results. This is only triggered by the following `EVAL` statement. Lets assume the user wants to drill-down on attribute 'state'. This means, she has to modify the query. She could do this as follows:

```
REDEFINE SESSION $s
  INSERT $f/customer$ AFTER $f/state on rows

$ret = eval($s)
```

   Impact: Modifications performed by the user on the reporting front-end are directly translated into statements of the query language. It is not necessary anymore to reissue the entire query.

7. AQL should allow to create query subscriptions (aka ative queries).

   Example statement:

```
define session $s as
       for $f in //profits
       group by $f/state on rows,
       $f/product on columns
       return as mdview

define function notify(
      $res as $s/result,
      $metadata as $s/metadata
   )
   ON $s CHANGED
   {
       (: code to handle query result $res :)
   }
```

   The function `notify` is called whenever the subscribed query produces a different result. This is either the case if the underlying data is changed or the query session gets modifed by a `redefine` statement. This mechanism can easily be used by the client software to redraw the screen: everything that remains to be done is to call `redraw`

whenever `notify` is called. In that case no explicit call to `eval` is necessary anymore:

```
define function notify(
      $res as $s/result,
      $metadata as $s/metadata
   )
   ON $s CHANGED
   {
       call redraw_result_screen($res, $metadata)
   }
```

Impact: This statement facilitates implementation. In addition, this feature greatly facilitates active warehousing and monitoring applications.

6 Conclusion

Despite all efforts, database vendors are not making the impact on the OLAP market that they could have. BI vendors such as SAP, Cognos, or i-TV-T build there own engines on top of DB products, thereby replicating a great deal of DB functionality and only using the very basic SQL 92 functionality (joins, group by and nested queries). The reason is that DBMS vendors are still overlooking some of the fundamental deficiencies of SQL and the relational model. The gap is widening and more and more stuff is added to OLAP engines that should ideally be implemented inside the DBMS. This paper has explored the gap between OLAP and SQL from a vendor point of view. Our contribution is threefold: First, we presented the gap vendors are confronted with when building reporting engines on top of current DBMS technology. Second, we showed how this gap can be bridged by a commerical OLAP engine, i-TV-T's BTell product. Third, we presented a wish list on how to extend DBMS to close the gap, i.e., how to better support OLAP and reporting functionality in the future.

We hope that our work revives discussions in the research community on the suitability of SQL for modern OLAP systems.

References

[1] K. Beyer, D. Chamberlin, L. Colby, F. Ozcan, H. Pirahesh, and Y. Xu. Extending XQuery for Analytics. In *ACM SIGMOD*, 2005 (to appear).

[2] M. J. Carey and D. Kossmann. Processing Top N and Bottom N Queries. *IEEE Data Engineering Bulletin*, 20(3):12–19, 1997.

[3] C. Cunningham, G. Graefe, and C. A. Galindo-Legaria. PIVOT and UNPIVOT: Optimization and Execution Strategies in an RDBMS. In *VLDB*, pages 998–1009, 2004.

[4] J. Doppelhammer, T. Höppler, A. Kemper, and D. Kossmann. Database Performance in the Real World: TPC-D and SAP R/3. In *ACM SIGMOD*, pages 123–134, 1997.

[5] C. D. French. "One Size Fits All" Database Architectures Do Not Work For DSS. In *ACM SIGMOD*, pages 449–450, 1995.

[6] C. D. French. Teaching an OLTP Database Kernel Advanced Data Warehousing Techniques. In *IEEE ICDE*, pages 194–198, 1997.

[7] F. Geerts, H. Mannila, and E. Terzi. Relational Link-based Ranking. In *VLDB*, pages 552–563, 2004.

[8] G. Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE TKDE*, 6(1):120–135, 1994.

[9] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Total. In *IEEE ICDE*, pages 152–159, 1996.

[10] H. Gupta, V. Harinarayan, A. Rajaraman, and J. D. Ullman. Index Selection for OLAP. In *IEEE ICDE*, pages 208–219, 1997.

[11] C. A. Hurtado and A. O. Mendelzon. Reasoning about Summarizability in Heterogeneous Multidimensional Schemas. In *IEEE ICDT*, pages 375–389, 2001.

[12] C. A. Hurtado and A. O. Mendelzon. OLAP Dimension Constraints. In *ACM PODS*, pages 169–179, 2002.

[13] I. F. Ilyas, R. Shah, W. G. Aref, J. S. Vitter, and A. K. Elmagarmid. Rank-aware Query Optimization. In *ACM SIGMOD*, pages 203–214, 2004.

[14] ISO/IEC. SQL 1999. *9075-1:1999*.

[15] R. Kimball and K. Strehlo. Why Decision Support Fails and How To Fix It. *ACM SIGMOD Record*, 24(3):92–97, 1995.

[16] H.-J. Lenz and A. Shoshani. Summarizability in OLAP and Statistical Data Bases. In *SSDBM*, pages 132–143, 1997.

[17] R. MacNicol and B. French. Sybase IQ Multiplex - Designed For Analytics. In *VLDB*, pages 1227–1230, 2004.

[18] M. Rafanelli and A. Shoshani. STORM: A Statistical Object Representation Model. In *SSDBM*, pages 14–29, 1990.

[19] A. Witkowski, S. Bellamkonda, T. Bozkaya, G. Dorman, N. Folkert, A. Gupta, L. Shen, and S. Subramanian. Spreadsheets in RDBMS for OLAP. In *ACM SIGMOD*, pages 52–63, 2003.

[20] J. Zhou and K. A. Ross. Buffering Database Operations for Enhanced Instruction Cache Performance. In *ACM SIGMOD*, pages 191–202, 2004.