

# Querying data warehouses efficiently using the Bitmap Join Index OLAP Tool

**Anderson Chaves Carniel**

São Paulo Federal Institute of Education, Science and Technology, IFSP, Salto Campus,  
Salto, SP, Brazil, 13.320-271  
*accarniel@gmail.com*

and

**Thiago Luís Lopes Siqueira**

São Paulo Federal Institute of Education, Science and Technology, IFSP, São Carlos Campus,  
São Carlos, SP, Brazil, 13.565-905  
Federal University of São Carlos, UFSCar, Computer Science Department,  
São Carlos, SP, Brazil, 13.565-905  
*prof.thiago@ifsp.edu.br*

## Abstract

Data warehouse and OLAP are core aspects of business intelligence environments, since the former store integrated and time-variant data, while the latter enables multidimensional queries, visualization and analysis. The bitmap join index has been recognized as an efficient mechanism to speed up queries over data warehouses. However, existing OLAP tools does not use strictly this index to improve the performance of query processing. In this paper, we introduce the BJIn OLAP Tool to efficiently perform OLAP queries over data warehouses, such as roll-up, drill-down, slice-and-dice and pivoting, by employing the bitmap join index. The BJIn OLAP Tool was implemented and tested through a performance evaluation to assess its efficiency and to corroborate the feasibility of adopting the bitmap join index to execute OLAP queries. The performance results reported that our BJIn OLAP Tool provided a performance gain that ranged from 31% up to 97% if compared to existing solutions regarding the query processing. Our tool has proven not only to efficiently process queries, but also to process OLAP operations on the server and client sides, for different volumes of data and taking into account different operating systems. Besides, it provides a reasonable use of the main memory and enables new rows to be appended to bitmap join indices.

**Keywords:** bitmap join index, OLAP, drill-down, roll-up, slice-and-dice, pivoting

## 1 Introduction

Business Intelligence solutions (BI) are widely adopted by management sectors of enterprises to aid processing, analysis and interpretation of their data, aiming at positively impacting strategy, tactics, and operations [1]. Data warehouse (DW) and Online Analytical Processing (OLAP) are core aspects of BI environments [2]. The DW is a subject-oriented, integrated, time-variant and non-volatile dimensional database [3], while OLAP provides tools to perform multidimensional queries over the DW and to support visualization and analysis of the DW [4]. Combining both the DW and OLAP enables a better monitoring of business. Therefore, many OLAP tools have been developed to end users visualize and manipulate multidimensional data, such as Oracle BI [5] and Mondrian [6].

The multidimensional operations commonly supported by OLAP tools are *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* [1][7]. They force the OLAP tool to challenge performance issues, since costly joins among huge fact and dimension tables as well as grouping operations are required together with predicates that filter the results. Aiming at reducing the query response time in DW, well-known methods as vertical fragmentation [8], view materialization [9][10][11][12] and indices [13][14] were proposed. However, none of the studied OLAP tools has investigated the

feasibility of adopting exclusively the bitmap join index [13][15] to improve the query processing performance in DW, although this index avoids costly join operations.

Providing such investigation is one of the contributions of this paper. In addition we introduce the Bitmap Join Index OLAP Tool (BJIn OLAP Tool) to efficiently perform *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* OLAP operations, as our main contribution. Our tool has proven to efficiently process these operations both on the server and client sides, for different volumes of data and also with portability for different operating systems. In addition, it provides a reasonable use of the main memory and enables new rows to be appended to bitmap join indices.

This paper extends a previous work [16], which was published and presented in *CLEI'2011 - XXXVII Conferencia Latinoamericana de Informatica*, in Quito, Ecuador. Furthermore, this paper extends another previous work [17]. We highlight several unpublished subjects addressed in this paper, as follows. We present an extended description of the BJIn OLAP Tool that comprises the system architecture, all the available operations (building, querying, appending new rows to and dropping bitmap join indices) and some implementation details. Moreover, the experimental evaluation discusses novel performance tests concerning the interface (client), the append operation and the portability. The remaining of this paper is organized as follows. Section 2 summarizes the technical background necessary to comprehend this paper. Section 3 introduces the BJIn OLAP Tool. Section 4 discusses the experimental results. Section 5 surveys related work. Finally, Section 6 concludes the paper and addresses future work.

## 2 Technical Background

In this section, the technical background necessary to comprehend the paper is summarized. In Section 2.1, data warehouse and OLAP concepts, applications and examples are addressed. In Section 2.2, methods to provide an efficient query processing over DW are described. Finally, Section 2.3 details how to append new rows.

### 2.1 Data warehouse and OLAP

Fig. 1 shows a star schema representing a retail application, which is derived from the Star Schema Benchmark (SSB) [18]. *Lineorder* is the fact table that measures sales and orders, while *Customer*, *Supplier*, *Part* and *Date* are dimension tables that redundantly store descriptive attributes that categorize the facts. These dimension tables are referenced by the fact table through foreign keys. In addition, the dimension tables hold hierarchies that enable data aggregation according to different granularity levels, such as  $(c\_region) \preceq (c\_nation) \preceq (c\_city) \preceq (c\_address)$ , which is held by the dimension table *Customer*, and  $(p\_mfgr) \preceq (p\_category) \preceq (p\_brand1) \preceq (p\_partkey)$  which is held by the dimension table *Part*. Considering the mentioned hierarchy in the dimension table *Customer*, *c\_region* is the highest granularity level, while *c\_address* is the lowest granularity level. According to [11],  $Q_1 \preceq Q_2$  if, and only if it is possible to answer  $Q_1$  using just the results of  $Q_2$ , and  $Q_1 \neq Q_2$ . Therefore, it is possible to find out the revenue in a given nation by aggregating the results of the cities inside that nation, for example. Finally, an alternative to the star schema is the snowflake schema that normalizes the hierarchies. However, the snowflake schema introduces additional costly join operations among dimension tables in order to process queries [19].

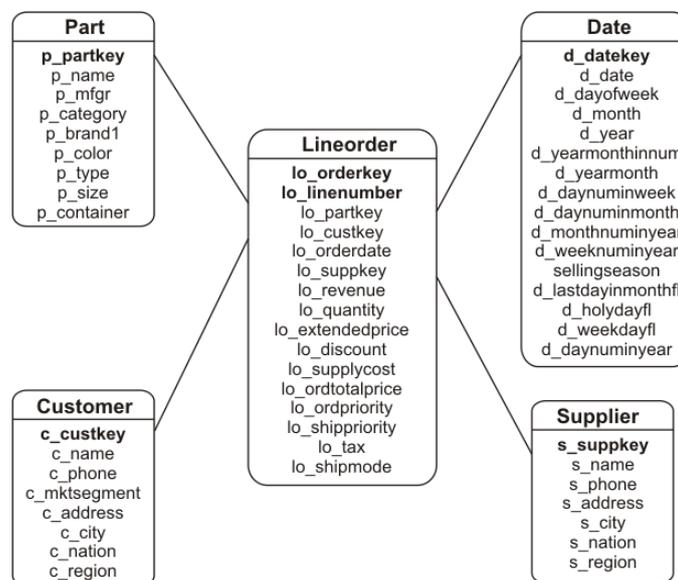
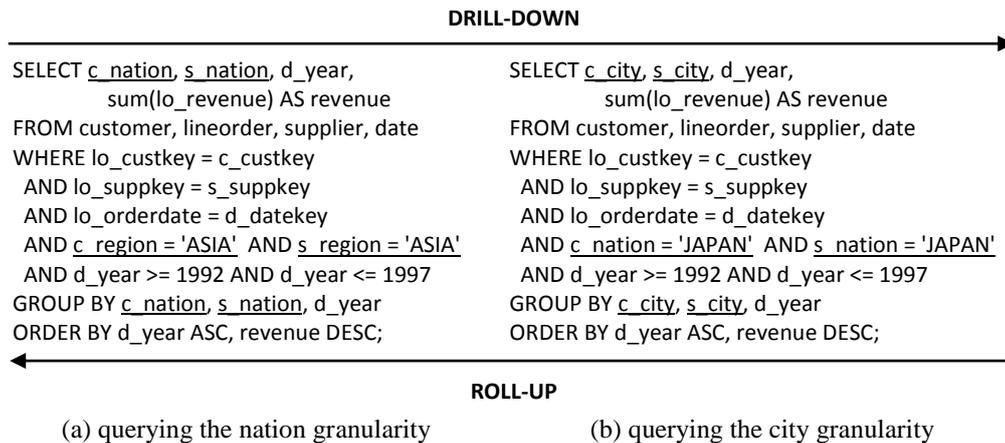


Fig. 1: A star schema for a DW of a retail application [13].

*Drill-down* and *roll-up* OLAP operations depend on hierarchies [1]. A *drill-down* operation decomposes fact data to lower levels of a hierarchy, then increasing data details. Inversely, a *roll-up* operation aggregates fact data to upper levels of a hierarchy, then summarizing data [7]. Fig. 2 shows examples of these operations adapted from [18], using existing hierarchies held by the dimension tables *Customer* and *Supplier*. Considering that the user firstly issued the query of Fig. 2a and later issued the query of Fig. 2b, there was a *drill-down* operation based on both  $(c\_nation) \preceq (c\_city)$  and  $(s\_nation) \preceq (s\_city)$ . On the other hand, if the user had issued the queries inversely, there was a *roll-up* operation based also on those mentioned hierarchies. The underlined attributes in Fig. 2 highlight these operations.

Both the queries of Fig. 2a and Fig. 2b exemplify the *slice-and-dice* operation, which consists of applying filters to the resulting data [7], such as “*c\_region = 'ASIA' AND s\_region = 'ASIA' AND d\_year >= 1992 AND d\_year <= 1997*”, shown in Fig. 2a. Finally, the *pivoting* operation enables reordering results by switching the axis for columns and rows [7]. Fig. 3a shows the results for the query of Fig. 2a, whose column *d\_year* was pivoted to be a row, providing the results of Fig. 3b. The representation of results in Fig. 3b is also known as a *cross table* [6].

OLAP tools support OLAP operations that are executed over the DW, such as *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* and enable multidimensional visualization and analysis [1]. Mostly, the data cube is accessed through Multidimensional Expressions (MDX) [20]. For instance, Mondrian is an open source OLAP server that comprises these features and reports query results on Java Server Pages by rendering the cross table employing JPivot [6] and synchronous requests that are sent to the server [21]. In order to enable OLAP operations, Mondrian requires the data cube definition in XML format describing the DW schema, e.g. fact and dimension tables, hierarchies and measures. The user may execute the Mondrian Schema Workbench and provide the proper inputs to generate the XML document containing the description of the DW. Otherwise, the user may execute any XML editor. This file ensures the correct access to tables, attributes and hierarchies when executing queries on Mondrian. Regarding query execution, the user types the query using MDX accessing the Mondrian interface. The MDX code is translated by Mondrian to SQL to access the database management system (DBMS), execute the query and finally retrieve the answers. The result set is then rendered in *cross tables* and charts and presented to the user, and then *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* operations are enabled. Conversely, recent web applications are adopting asynchronous requests based on Ajax and JSON (JavaScript Object Notation) [26]. Therefore we decided to adopt asynchronous requests to develop the BJIIn OLAP Tool.



**Fig. 2:** Roll-up and drill-down operations

c_nation	s_nation	d_year	revenue
CHINA	CHINA	1992	5587028770
CHINA	CHINA	1993	5241310984
CHINA	CHINA	1994	5452596836
CHINA	CHINA	1995	5335157374
CHINA	CHINA	1996	5436388668
CHINA	CHINA	1997	5281192823
CHINA	INDIA	1992	4912422786
...	...	...	...

(a) query results

<b>1992</b>	c_nation	s_nation	revenue
	CHINA	CHINA	5587028770
	CHINA	INDIA	4912422786
	CHINA	INDONESIA	5259493502
	CHINA	JAPAN	5040640767
	CHINA	VIETNAM	4437447402
	INDIA	CHINA	4698961607
	INDIA	INDIA	4156866160
...	...	...	...

(b) cross table with column *d\_year* pivoted

**Fig. 3:** The original query results and the pivoted query results reorganized with year as a row

## 2.2 Improving query processing performance over data warehouses

The costly method to process a query over a DW is to perform the *star-join*, by joining all tables of the star schema and then perform filters, groupings and sorting. This strategy provides prohibitive query response times, as discussed in Section 4. On the other hand, the methods discussed in this section can improve the query processing performance over DW, and are employed in the performance evaluation presented in Section 4.

Some methods store pre-computed data into tables after performing some operations. A *vertically fragmented view* [8] maintains the minimum set of columns of the star schema that are necessary to answer a given query. For instance, the table shown in Fig. 4a has the minimum set of columns to provide the answer to the query shown in Fig. 2a. Note that all essential joins were computed when composing the view, by issuing  $\Pi_{c\_region, s\_region, c\_nation, s\_nation, d\_year, lo\_revenue} (Customer \bowtie Lineorder \bowtie Supplier \bowtie Date)$ . Therefore, this view can be stored aiming at improving the query processing performance, since joins are avoided and only filters and groupings need to be computed to retrieve the query answer.

On the other hand, *materialized views* [9][11][12] pre-compute the DW information that can be used to answer queries that are frequently issued. A materialized view is built by creating a table to report pre-computed data from a fact table that was joined to dimension tables, and whose measures were aggregated. Since a materialized view stores pre-computed aggregated data, processing a query avoids joins and groupings, and drastically reduces the number of rows then benefiting the filters. For instance Fig. 5a depicts a materialized view created as  $\Pi_{c\_nation, s\_nation, d\_year} G_{SUM}(lo\_revenue) (\Pi_{c\_region, s\_region, c\_nation, s\_nation, d\_year, lo\_revenue} (Customer \bowtie Lineorder \bowtie Supplier \bowtie Date))$  to efficiently answer the query shown in Fig. 2a. This view can be stored aiming at improving the query processing performance, since joins and groupings are avoided, and only filters need to be computed to retrieve the query answer. The reduced set of rows benefits the performance of filtering. Finally, both vertically fragmented views and materialized views can be applied to OLAP tools to enhance the query processing performance over DW. For example, materialized views can be applied to Mondrian, i.e., aggregate tables according to Mondrian's terminology.

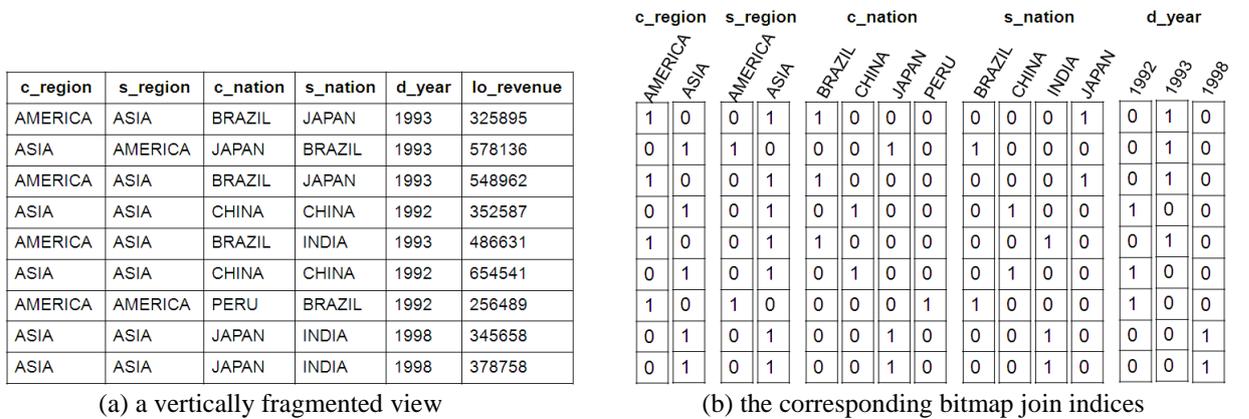


Fig. 4: Vertically fragmented view and Bitmap join indices

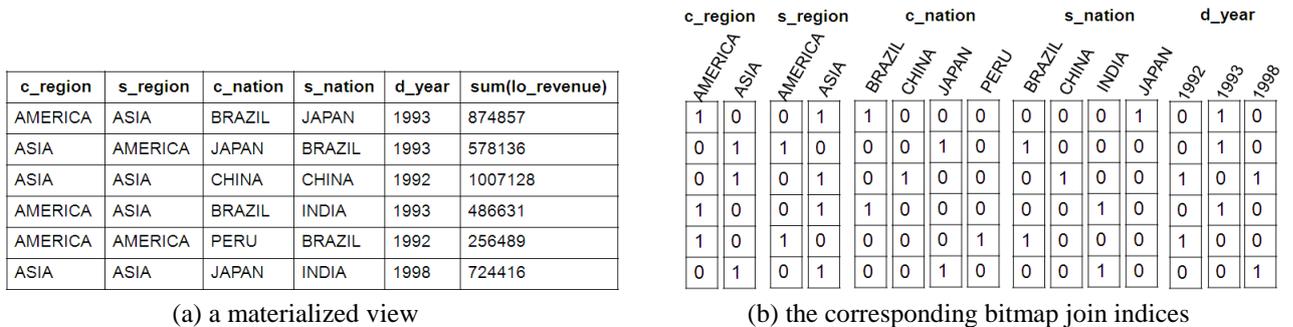


Fig. 5: Materialized view and Bitmap join indices

Indices are an alternative to storing pre-computed data. The *bitmap index* [14] builds one bit-vector to each distinct value  $v$  of the indexed attribute  $A$ . The attribute cardinality,  $|A|$ , is the number of distinct values of  $A$  and

determines the quantity of existing bit-vectors. All bit-vectors have as many bits as the number of rows found in the indexed table. If for the  $i$ -th record of the table we have that  $A = v$ , then the  $i$ -th bit of the bit-vector built for value  $v$  is set to 1. Otherwise, the bit is set to 0. Suppose that the attribute  $d\_year$  Fig. 3a is indexed by a *bitmap index*. The cardinality  $|d\_year|$  is 6, resulting in six bit-vectors, each one of them associated to one of the values 1992, 1993, 1994, 1995, 1996 and 1997. For instance, the bit-vector for  $d\_year=1992$  is 1000001, denoting the existence of  $d\_year=1992$  in the first and seventh rows. The main advantage of processing queries using a bitmap index is the CPU efficiency of bitwise operations (i.e. AND, OR, XOR, NOT) [22]. For instance, to query “ $d\_year \geq 1992$  AND  $d\_year \leq 1995$ ”, there are bit-wise OR operations among the bit-vectors for the values involved, i.e., 1000001 OR 0100000 OR 0010000 OR 0001000. The result, 1111001, excludes the fifth and sixth rows from the result set. High cardinality attributes may impair the performance of the *bitmap index*, but binning, encoding and compression techniques minimize these losses [14]. Currently, the FastBit is an efficient open source implementation of the bitmap index [23].

Besides, a *bitmap join index* [13] can be created on the attribute  $B$  of a dimension table in order to indicate the set of rows in the fact table to be joined with a certain value of  $B$ . Therefore, each bit determines the rows of the fact table where a given value of  $B$  exists. As already mentioned, Fig. 4a shows the table obtained from applying  $\Pi_{c\_region, s\_region, c\_nation, s\_nation, d\_year, lo\_revenue} (Customer \bowtie Lineorder \bowtie Supplier \bowtie Date)$  on the star schema shown in Fig. 1 to answer the query of Fig. 2a. This table is the fact table *Lineorder* joined with the dimensions *Customer*, *Supplier* and *Date*. In Fig. 4b, bitmap join indices were built on attributes  $c\_region$ ,  $s\_region$ ,  $c\_nation$ ,  $s\_nation$  and  $d\_year$  to improve the performance when processing the query shown in Fig. 2a. As a result, these indices indicate the rows of the table shown in Fig. 4a where a given value occurs. For instance,  $d\_year = 1998$  occurs in the 8<sup>th</sup> and 9<sup>th</sup> rows of the table shown in Fig. 4a. Joins among huge DW tables are necessary only once to build the bitmap join index. After the index is built, the queries can be processed by accessing the index, avoiding costly joins among the tables of the DW. Bitmap join indices built over materialized views, as shown in Fig. 5b, are capable of processing queries even more efficiently than those built over vertically fragmented views, because the former maintain aggregated data (and a reduced data volume) while the latter does not. Although the bitmap join index improves the query processing over DW, none of the OLAP tools investigated in Section 5 adopted exclusively this index to process OLAP queries such as *drill-down*, *roll-up*, *slice-and-dice* and *pivoting*.

### 2.3 Appending rows to data warehouses, views and indices

In this section, issues related to appending rows to vertically fragmented views, materialized views and bitmap join indices are discussed [7][19][24][25], since they are essential to understand the append operation developed for the BJJn OLAP Tool. New rows are appended to DW on a regular time cycle, e.g. daily, weekly or monthly. Suppose that the three rows shown in Fig. 6 are appended to the DW depicted in Fig. 1. Consider, also, that  $lo\_suppkey=715$  references a supplier located in Brazil ( $t1$ ),  $lo\_custkey=22851$  references a customer located in Japan ( $t1$ ), and  $lo\_orderdate=871$  occurred in 1993 ( $t1$  and  $t3$ ).

$t1$

<u>lo_orderkey</u>	<u>lo_linenumbr</u>	<u>lo_partkey</u>	<u>lo_custkey</u>	<u>lo_orderdate</u>	<u>lo_suppkey</u>	<u>lo_revenue</u>	...	<u>lo_shipmode</u>
78171831	15	1	22851	871	715	7890		AIR

$t2$

<u>c_custkey</u>	<u>c_name</u>	<u>c_phone</u>	<u>c_mktsegment</u>	<u>c_address</u>	<u>c_city</u>	<u>c_nation</u>	<u>c_region</u>
30001	EC7000	+59388112278	AUTOMOBILE	EC18930	QUITO	ECUADOR	AMERICA

$t3$

<u>lo_orderkey</u>	<u>lo_linenumbr</u>	<u>lo_partkey</u>	<u>lo_custkey</u>	<u>lo_orderdate</u>	<u>lo_suppkey</u>	<u>lo_revenue</u>	...	<u>lo_shipmode</u>
78171831	16	1	30001	871	715	92910		RAIL

**Fig. 6:** Three rows to be appended to the DW depicted in Fig. 1

Firstly, the row  $t1$  is inserted in the fact table *Lineorder*. Considering the vertically fragmented view of Fig. 4a, a new row is appended, as {'ASIA', 'AMERICA', 'JAPAN', 'BRAZIL', 1993, 7890}. In Fig. 4b, the bit-vector for  $c\_region='AMERICA'$  has a bit 0 appended, while the bit-vector for  $c\_region='ASIA'$  has a bit 1 appended, and the other attributes' bit-vectors are similarly modified. As for the materialized view of Fig. 5a, the second row should be modified to comprise the value 586026 (i.e. 578136+7890) for the attribute  $sum(lo\_revenue)$ . Finally, no one of the bit-vectors in Fig. 5b need to be modified, since any new row was inserted in the corresponding materialized view of Fig. 5a. However, if the attribute  $sum(lo\_revenue)$  was indexed, the bit-vector for value 578136 would be modified by replacing a bit 1 by a bit 0 in the second row, i.e. the bit-vector would be modified from 010000000 to 000000000. In addition, a bit-vector for value 586026 (i.e. 578136+7890) would be created and have a bit 1 in the

second row and zeroes in the remaining rows, i.e. 01000000. Clearly, the insertion of *t1* in the DW reveals several challenges for maintaining vertically fragmented views, bitmap join indices or materialized views, which are necessary to speed up the query processing.

Secondly, the row *t2* is inserted in the dimension table *Customer* and does not affect the vertically fragmented view, the bitmap join indices or the materialized views since these are exclusively associated to facts. And thirdly, *t3* is inserted in the fact table *Lineorder*. As a result, the vertically fragmented view of Fig. 4a has a new row appended, as {'AMERICA', 'AMERICA', 'ECUADOR', 'BRAZIL', 1993, 92910}. As for the bitmap join indices shown in Fig. 4b, it is necessary to: (i) append a bit 1 in the bit-vector for *c\_region*='AMERICA' and append a bit 0 in the bit-vector for *c\_region*='ASIA'; (ii) append a bit 1 in the bit-vector for *s\_region*='AMERICA' and append a bit 0 in the bit-vector for *s\_region*='ASIA'; (iii) build a new bit-vector for *c\_nation*='ECUADOR', with a bit 1 in the last row and bits zeroes in the remaining rows; (iv) append a bit 1 in the bit-vector for *s\_nation*='BRAZIL' and append a bit 0 in the remaining bit-vectors of this attribute; and (v) append a bit 1 in the bit-vector for *d\_year*=1993 and append a bit 0 in the remaining bit-vectors of this attribute. As for the materialized view shown in Fig. 5a, a new row composed of {'AMERICA', 'AMERICA', 'ECUADOR', 'BRAZIL', 1993, 92910} is appended. If the attribute *sum(lo\_revenue)* was indexed, a new bit-vector for value 92910 would be built with a bit 1 in the last row and zeroes in the remaining rows. Finally, the bitmap join index on *c\_nation* in Fig. 5b earns a new bit-vector for value 'ECUADOR', while the other bit-vectors are modified similarly to those from Fig. 4b, as discussed. Clearly, the insertion of *t2* and *t3* in the DW reveals even more challenges for maintaining vertically fragmented views, bitmap join indices or materialized views aiming at speeding up the query processing. Particularly, the bitmap join indices required the creation of more bit-vectors, since a new customer was inserted in the *Customer* dimension table. The BJI OLAP Tool supports appending new rows to the DW similarly to the insertion of *t1*, *t2* and *t3*, as detailed in Section 3.3.

### 3 The Bitmap Join Index OLAP Tool

The architecture of the Bitmap Join Index OLAP Tool (BJIn OLAP Tool) is shown in Fig. 7. The BJI OLAP Tool was developed as an open source OLAP server written in Java that accesses bitmap join indices to speed up the OLAP operations *drill-down*, *roll-up*, *slice-and-dice* and *pivoting*. On the server side, our tool operates both the DBMS and the FastBit in order to build the indices, to issue queries over them and to append new rows to them. The queries are submitted by the client to the server, and the latter accesses strictly the indices to provide the answer rapidly with high performance. On the client side, the user interacts with our tool through Java Server Pages, submits queries and analyzes multidimensional data that are rendered on cross tables and charts produced by the Open Ajax Toolkit Framework. Whenever a cross table is modified by the user to produce another view, the corresponding chart is refreshed and synchronized with the cross table, and vice-versa. Some implemented facilities aid users to interact, i.e. the visualization of the data cube as a tree to select attributes to index and highlight and auto complete the query string to match the proper syntax. Finally, other utilities manipulate internal files to maintain logs, access privileges, configuration parameters, metadata and parsing.

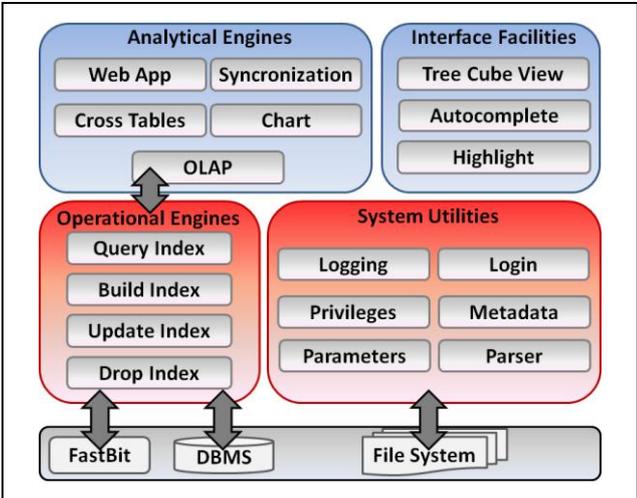


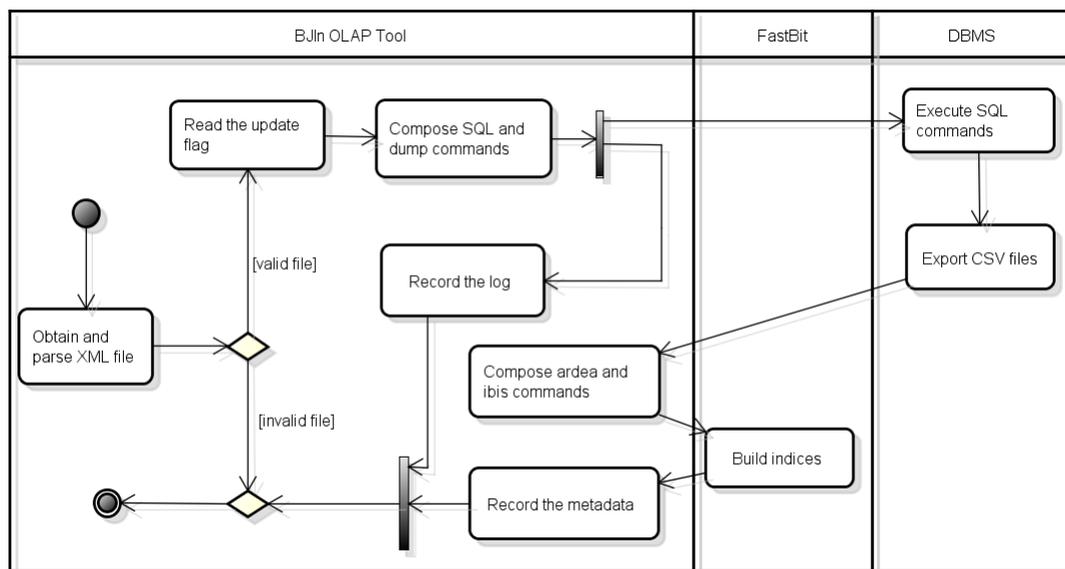
Fig. 7: The architecture of the BJI OLAP Tool

Sections 3.1, 3.2, 3.3 and 3.4 describe building, query processing, data appending and drop operations over bitmap join indices using the BJIIn OLAP Tool, respectively. Section 3.5 details additional features. We encourage the reader to access the BJIIn OLAP Tool Portal at <http://gbd.dc.ufscar.br/bjinolap>.

### 3.1 Building the bitmap join indices

Before using the BJIIn OLAP Tool to build bitmap join indices, the user should execute the Mondrian Schema Workbench to specify the attributes to be indexed, as well as dimension and fact tables, measures, and hierarchies that exist in the DW schema. The Workbench validates these inputs by checking the DW schema, i.e., by accessing the DBMS, assuring that the DW was properly described by the user. If the validation is successful, the Workbench generates a XML document that stores all the DW schema specification and the attributes to be indexed. The reuse of Mondrian Schema Workbench promotes the interoperability between Mondrian OLAP Server and the BJIIn OLAP Tool, since the produced XML document can be used by both. While Mondrian reads the document to compose a data cube, our tool parses it in order to build the bitmap join indices on the specified attributes. However, the use of the Mondrian Schema Workbench does not impose a restriction, because another XML editor could be employed instead, since the syntax and tags remains the same.

After specifying all parameters, the user logs in the BJIIn OLAP Tool, uploads the corresponding XML document and sets or unsets the append flag, which determines if the indices shall support new rows to be appended or not (as detailed in Section 3.3). Thereafter, the UML activity diagram shown in Fig. 8 models the whole process of how to build bitmap join indices using our tool. Once uploaded, the XML document is parsed by the BJIIn OLAP Tool, which issues SQL and dump commands on the DBMS in order to compute joins and build a temporary table. This table is dumped to a set of CSV files (comma-separated values) that are stored into the BJIIn OLAP directory. Then, the BJIIn OLAP Tool issues *ardea* and *ibis* commands to the FastBit. While the former reads CSV files to store data into the FastBit binary format, the latter effectively builds the bitmap index and stores it into the directory. Finally, the BJIIn OLAP Tool records metadata that fully specifies the index, e.g. the names and types of the indexed columns, aliases and the available OLAP operations for that index. The log recording starts after the composition of SQL and dump commands and finishes after metadata are recorded. The log file is detailed in Section 3.5 and maintains a complete description of the building operation.



**Fig. 8:** Building bitmap join indices using the BJIIn OLAP Tool

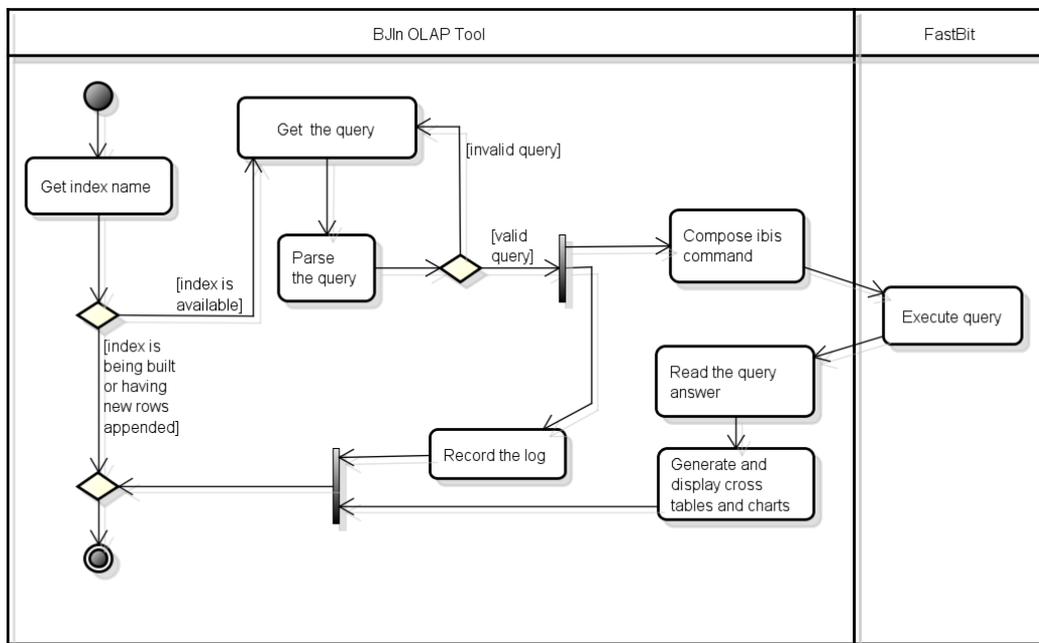
For instance, suppose that *c\_region*, *s\_region*, *c\_nation*, *s\_nation*, *d\_year* and *lo\_revenue* from the DW depicted in Fig. 1 were specified by the user to be indexed. They involve four different tables to be joined: *Customer*, *Lineorder*, *Date* and *Supplier*. Therefore, the temporary table to be created by the SQL commands is exactly the one shown in Fig. 4a. Then, the bitmap join indices are created on the attributes of this temporary table. Note that indexing such attributes would enable *roll-up* or *drill-down* operations considering that  $(c\_region) \preceq (c\_nation)$ . However, in order to enable *roll-up* or *drill-down* operations along the entire hierarchies  $(c\_region) \preceq (c\_nation) \preceq (c\_city) \preceq (c\_address)$  and  $(s\_region) \preceq (s\_nation) \preceq (s\_city) \preceq (s\_address)$ , the user may have firstly specified each one of these attributes on Mondrian Schema Workbench.

A relevant remark is that the user can alternatively create bitmap join indices over materialized views, similarly to Fig. 5. Considering the activity diagram shown in Fig. 8, in addition to upload the XML document and set or unset the append flag, the user may check the option to build indices over the materialized view and then specify the attributes to be indexed. To specify these attributes, the user marks them on the BJJn OLAP Tool interface (i.e. the Tree Cube View component, shown in Fig. 7). Then, the DBMS builds a temporary table that corresponds to the materialized view containing the specified attributes. The creation of the materialized view causes an overhead because it requires data aggregation. On the other hand, the index has a reduced data volume and a better performance on query processing.

The BJJn OLAP Tool provides mechanisms to avoid, treat and report errors during the tasks to build bitmap join indices. The tool refuses the upload of any invalid XML documents or indices that are homonyms. All attributes have internal aliases that avoid ambiguity. Also, our tool limits the data volume to be manipulated by the FastBit, avoiding memory leaks.

### 3.2 Query processing

Whenever the user builds an index, its metadata is recorded and then the index becomes available to be queried. The UML activity diagram shown in Fig. 9 models the whole process of how to process queries using our tool. Initially, the user chooses the index to be used among all available indices, and types the desired query. The BJJn OLAP Tool parses the query and writes the proper *ibis* command containing the query and the chosen index, and submits it to the FastBit. Then, the FastBit accesses the index and processes the query. After processing the query, the FastBit writes a CSV file containing the query results. The BJJn OLAP Tool reads this CSV file to build the cross table and render it on Java Server Pages, which are displayed to the user. Additionally, charts are displayed to depict the same results of the cross table.



**Fig. 9:** Issuing queries to be processed by the BJJn OLAP Tool

After a query execution, rather than typing another query the user is able to perform OLAP operations as follows. Once the results were displayed, the *pivoting* operation is allowed. All the user needs to do is to drag and drop columns or rows to switch the axis of the cross table. This operation is computed on the client side, and therefore was not shown in Fig. 9. *Drill-down* and *roll-up* operations are also allowed for the user if the requested attributes were indexed and if there is at least one hierarchy involved in the previous query. For instance, if the previous query involves the *s\_nation* attribute, a combo-box will enable the attribute *s\_region* for the *roll-up* operation, and the attributes *s\_city* and *s\_address* for the *drill-down* operation. The user then selects the operation and the attributes of interest in the combo-box. Furthermore, *roll-up* and *drill-down* operations are executed on the server side, and correspond to issuing a new query. However, since results of the previous query were cached by the server and

contain partial results of the new query, the performance is benefited. Every OLAP operation that the user applies to a cross table is also applied to synchronize the corresponding chart.

The query language used to compose queries is already defined by the FastBit and does not require joins or grouping clauses. The columns listed in the SELECT clause are used to aggregate results. Therefore, writing the query is a straightforward task for the user, since only SELECT-WHERE clauses need to be written. Furthermore, the *slice-and-dice* operation can be described as restrictions in the WHERE clause.

For instance, suppose that attributes *c\_region*, *s\_region*, *c\_nation*, *d\_year* and *lo\_revenue* were indexed and that the user issues the query “*SELECT c\_nation, s\_nation, d\_year, sum(lo\_revenue) WHERE AND c\_region = 'ASIA' AND s\_region = 'ASIA' AND d\_year >= 1992 AND d\_year <= 1997*”. The FROM clause is not necessary because the user had already selected the index to be queried. The WHERE clause has filters that define the *slice-and-dice* OLAP operation. To submit a *roll-up* operation on *c\_nation*, instead of typing another query, the user should simply select the attribute *s\_nation* in the combo-box. To perform a pivoting operation and switch the column *d\_year* to a row, the user should simply drag and drop this item. The results of these operations are automatically applied to the chart that depicts the corresponding cross table.

The BJIIn OLAP Tool provides mechanisms to avoid, treat and report errors during the tasks to process queries over bitmap join indices. Since the building and the data appending operations of bitmap join indices often spend several seconds (as discussed in Section 4), our tool does not enable queries over indices that are currently being built or having new rows appended. Besides, the tool refuses to issue queries that are syntactically wrong or that refer to attributes that are not indexed by the selected index. Ambiguity is avoided to issue and process queries, since all attributes have internal aliases. Moreover, any runtime error during the query execution is reported to the user. The log recording starts after parsing the query and finishes after displaying cross tables and charts. Finally, the OLAP operations of *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* are enabled only for attributes that were previously indexed and whose hierarchies were associated to the previous query.

### 3.3 Appending new rows to bitmap join indices

In the BJIIn OLAP Tool, the existing bitmap join indices support new rows to be appended if the user had set the append flag before building the indices (see Section 3.1 and Fig. 8). The UML activity diagram shown in Fig. 10 models the whole process of how to append new rows to existing bitmap join indices using the BJIIn OLAP Tool. Firstly, the user provides the name of the index that wishes to append new rows to. Then, our tool reads the metadata of the specified index and then issues SQL and dump commands on the DBMS. A temporary table whose rows must be appended to the indices is accessed, and its rows are dumped in CSV files. After dumping data, all the rows of the temporary table are deleted. Then, the BJIIn OLAP Tool composes *ardea* and *ibis* commands and issues them on FastBit using the cited CSV files to append the new rows and possibly create new bit-vectors. Finally, our tool updates the metadata file with the timestamp of the last row appended. The log recording starts after composing SQL and dump commands and finishes after recording the metadata.

In detail, whenever the user sets the append flag before building the indices, the BJIIn OLAP Tool automatically creates two main components:

- a temporary table with the same attributes of the indices; and
- a trigger that monitors if new rows are being appended to the fact table of the DW.

The temporary table remains stored as long as the corresponding indices exist (see more details in Section 3.4). Rows are inserted into this table whenever the trigger detects an insertion into the fact table, similarly to the rows *t1* and *t3* exemplified in Fig. 6. The trigger maps the foreign key values of each appended row to the corresponding values of attributes that were indexed. Also, the trigger inserts the mapped appended rows in the temporary table. This table then contains the set of rows to be appended to the bitmap join indices. The trigger has a sequence of tasks to be performed, independently of the DW schema. These tasks are detailed in Algorithm 1, whose parameters and local variables are described in Table 1.

Initially, the record to be inserted in the temporary table is empty (line 1). There is a loop to assure the processing of the following tasks for every row inserted in the fact table (lines 2 to 19). For each attribute of the inserted row, the values are mapped to the values that will be appended to the index (lines 3 to 14). Attributes that have foreign keys referencing the dimension tables (lines 4 to 12) are distinguished from those attributes that denote measures (line 13). Finally, the record with mapped values is inserted in the temporary table (lines 15 to 18).

To map the values of the attributes that reference the dimension tables through foreign keys (e.g. *lo\_custkey* in Fig. 1) to adequate values of the indexed attributes (e.g. *c\_region* and *c\_nation* in Fig. 1), each dimension table must be read (line 4). If one of the dimension tables store the given attribute (e.g. *Customer*), then it is necessary to compose the set of indexed attributes *C* (line 6). This set indicates all attributes in a given dimension table that also exist in the temporary table. For instance, if *Customer* is the dimension table and *temp* has the attributes of Fig. 4a, then *C* is assigned to  $\{c\_region, s\_region, c\_nation, s\_nation, d\_year, lo\_revenue\} \cap \{c\_custkey, c\_name,$

$c\_phone, c\_mktsegment, c\_address, c\_city, c\_nation, c\_region$  and therefore  $C = \{c\_nation, c\_region\}$ . Then, for each element of  $C$ , the value for that attribute is fetched in the dimension table and added to the record that maintains mapped values (lines 7 to 10). For example, if the row  $t1$  of Fig. 6 was inserted in the fact table, the values 'JAPAN' and 'ASIA' would be added to the record, since  $SELECT c\_nation FROM Customer WHERE c\_custkey = 22851$  and  $c\_region FROM Customer WHERE c\_custkey = 22851$  would be executed. After executing these steps for all dimension tables, the record would contain {'ASIA', 'AMERICA', 'JAPAN', 'BRAZIL', 1993}.

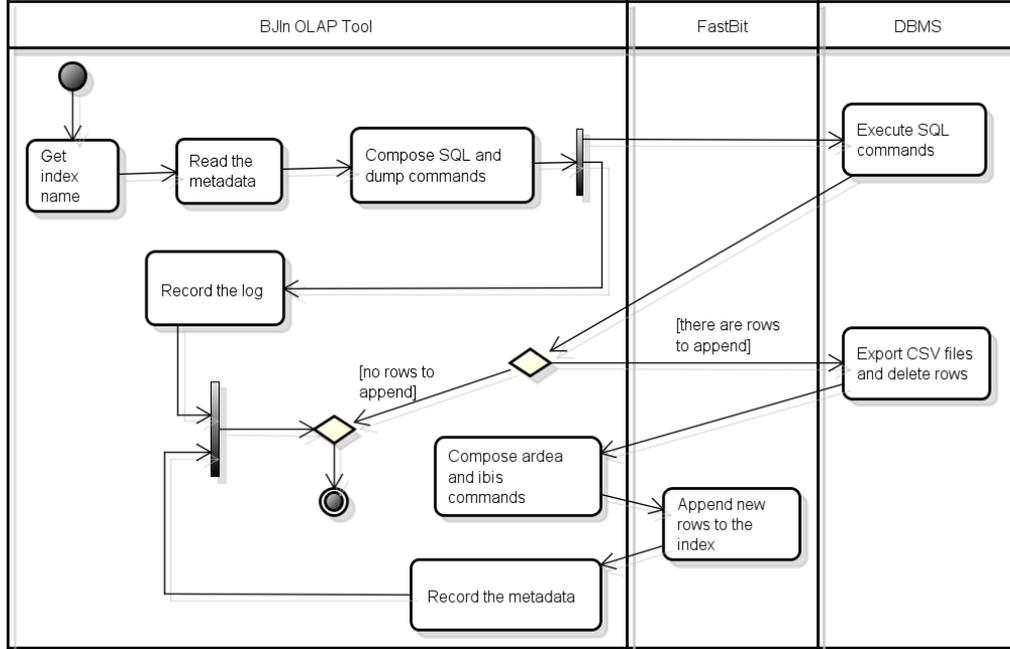


Fig. 10: Appending new rows to bitmap join indices with the BJIIn OLAP Tool

### Algorithm 1

**BitmapJoinIndexAppendRowsTrigger ( F, T,  $t_i$ ,  $a_j$ , D,  $d_k$ , M, temp )**

*Output: The temporary table containing data to be appended to the bitmap join indices.*

*Declarations: record, value, C, NEW*

```

01 record ← NULL
02 for each  $t_i$  in T
03   for each  $a_j$  in  $t_i$ 
04     for each  $d_k$  in D
05       if  $a_j \in d_k$  then
06          $C \leftarrow temp.getColumns() \cap d_k.getColumns()$ 
07         for each  $c_m$  in C
08           value ← execute_dbms(SELECT  $c_m$  FROM  $d_k$  WHERE  $d_k.pk = t_i.a_j$ )
09           record.add(value)
10         end-for
11       end-if
12     end-for
13   if  $a_j \in M$  then record.add(NEW. $a_j$ )
14 end-for
15 if record is not null then
16   execute_dbms(INSERT INTO temp VALUES record.getValues())
17   record ← NULL
18 end-if
19 end-for

```

Further, the values for the attributes that denote measures are added to the record (line 13). In the previous example, the value for  $lo\_revenue$  is added to the record, resulting in {'ASIA', 'AMERICA', 'JAPAN', 'BRAZIL', 1993, 7890}. As a result, the record store the mapped values. Finally, the insertion is performed in the temporary table using the mapped values from the record (lines 15 to 18), for example  $INSERT INTO temp VALUES ('ASIA', 'AMERICA', 'JAPAN', 'BRAZIL', 1993, 7890)$ . An important detail is to empty the record before processing another

row (line 17). Note that the BJIn OLAP Tool has a strict control over attributes that are homonyms, avoiding the ambiguity that could occur if attributes in distinct dimension tables had the same name (in line 5). One important remark is that Algorithm 1 supports only star schemas. To provide support for a snowflake schema, the BJIn OLAP Tool modifies line 8 to join the normalized tables and then fetch the attribute  $c_m$ . These tables are held by the set of dimension tables  $D$ .

**Table 1:** Parameters and local variables of Algorithm 1

Parameter or local variable	Description
$F$	the DW's fact table
$T$	the set of rows that are being inserted in the fact table $F$
$t_i$	a row from $T$
$a_j$	an attribute from $t_i$
$D$	the set of dimension tables
$d_k$	a dimension table from $D$
$M$	the set of measures in $F$
$temp$	the temporary table
$record$	a record of type $temp$
$value$	a value extracted from a given attribute
$C$	a set of attributes
$NEW$	the row that is being appended to $F$

Regarding the data appending to bitmap join indices using the BJIn OLAP Tool, we finally emphasize that:

- The append operation comprises the creation of new bit-vectors if necessary, similarly to the insertion of rows  $t1$  and  $t3$  in the fact table *Lineorder*, according to Fig. 6.
- Insertions in the fact table that are denied by the DBMS because they violate integrity or referential constraints are not considered by the BJIn OLAP Tool to append rows to the bitmap join indices.
- The replacement of values, similar to an UPDATE command of the SQL, is not supported. As exemplified in Section 2.2, this is the case for the bitmap join index built on the attribute *sum(lo\_revenue)* of Fig. 5a and the subsequent insertion of the row  $t1$  into *Lineorder*, according to Fig. 6.
- If bitmap join indices created over a materialized view requires the replacement of values, firstly the materialized view and later the bitmap join indices are rebuilt.
- The append operation is not automatic and requires the user intention because, similarly to a DW, this operation should be executed in batch and during a time window when the indices are unavailable to users.

### 3.4 Dropping the bitmap join indices

The UML activity diagram shown in Fig. 11 describes how to drop bitmap join indices using the BJIn OLAP Tool. Firstly, the user selects the index to be dropped and confirms the choice, because this operation is permanent and cannot be undone. Then, our tool checks if the user had set the append flag before building the index. If so, the DBMS is accessed and drops the corresponding temporary table and trigger. The files and directories concerning the index chosen are removed from the file system. The BJIn OLAP Tool provides mechanisms to avoid, treat and report errors during the tasks to drop bitmap join indices as follows. Only indices whose append flag were set require an access to the database. Besides, the metadata that describe the indices (Section 3.1) and the internal aliases for indices and their attributes avoid the deletion of indices that were not specified by the user.

### 3.5 Additional features

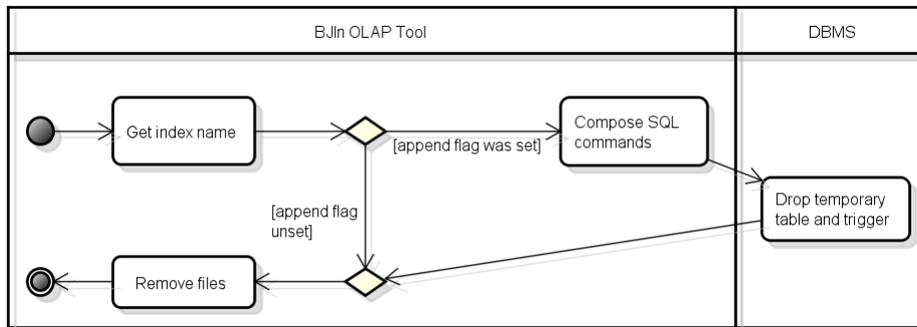
The operations involving bitmap join indices described in sections 3.1 to 3.4 require a previous authentication, i.e., the BJIn OLAP Tool only enables these operations if the user was previously identified and logged in. The privileges available to users are: *canUploadXml* to allow XML files to be uploaded by the user; *canCreateIndex* to allow the user to build indices; *canAppendRows* to allow indices to have new rows appended; *canDropIndex* to allow indices to be dropped by a given user; and *isSuperUser* to determine if the user is a superuser and therefore has no restrictions.

In order to configure the BJIn OLAP Tool to run properly in a given operating system, the user sets the parameters in the configuration file *config.properties*. The properties required are: *ibisPath* to indicate the directory where the *ibis* application was installed by the FastBit; *ardeaPath* to indicate the directory where the *ardea* application was installed by the FastBit; *url* to detail the Java Database Connection (JDBC) values; *driver* to specify

the JDBC class; and *bjinlapPath* to indicate the directory where the BJIn OLAP Tool will manipulate folders and files associated to the bitmap join indices.

Another relevant feature implemented in the BJIn OLAP Tool is the log, which records every command issued by the tool, e.g. DBMS and operating system commands. All the described operations detailed in the previous sections have their specific logs. Every runtime error is recorded. As a result, the log benefits debugging the software. This feature was implemented using the log4j library (<http://logging.apache.org>).

Currently, our tool is compatible to the operating systems Windows and Linux, to the DBMSs PostgreSQL, MySQL and IBM DB2®, and to the browsers Opera, Chrome, Firefox, IE8 and IE9. Regarding the Java Virtual Machine, its version 7 is compatible.



**Fig. 11:** Dropping bitmap join indices with the BJIn OLAP Tool

## 4 Experimental Evaluation

This section presents the experimental evaluation of the BJIn OLAP Tool, which was done by running performance tests. The results point out the remarkable performance of the BJIn OLAP Tool to process the following OLAP operations: *drill-down*, *roll-up* and *slice-and-dice*. We investigate the performance of our tool against the current technology of DBMS and against the Mondrian OLAP Server. Since the *pivoting* OLAP operation is performed on the client side, it was not evaluated in our tests.

In section 4.1 we detail the experimental setup used to execute the tests. The first test, in Section 4.2, compares the *slice-and-dice* query processing performance for the BJIn OLAP Tool, vertically fragmented views stored by the DBMS and Mondrian OLAP Server. Furthermore, storage requirements and attributes' cardinalities are addressed. In Section 4.3 we focus on the *drill-down* and *roll-up* query processing performance. Section 4.4 describes the results regarding a more voluminous DW and the use of materialized views and bitmap join indices built over these views. In Section 4.5, the performance of rendering interface components on the client side is assessed. Section 4.6 details the memory usage, while Section 4.7 evaluates the query processing performance and the portability issues. Finally, Section 4.8 focuses the cost of appending new rows to bitmap join indices using the BJIn OLAP Tool.

### 4.1 Experimental setup

Regarding the datasets, we used the Star Schema Benchmark (SSB) [18] to create two star schemas identical to that in Fig. 1. The DW1 dataset was loaded according to the SSB scale factor 1 and produced 6 million rows in the fact table, while the DW10 dataset was loaded with scale factor 10 and therefore was 10 times more voluminous than DW1. Both of them held attribute hierarchies such as  $(s\_region) \preceq (s\_nation) \preceq (s\_city) \preceq (s\_address)$  and  $(c\_region) \preceq (c\_nation) \preceq (c\_city) \preceq (c\_address)$ , then enabling the experimental evaluation of *drill-down* and *roll-up* operations. The DBMS automatically created B-trees to index the attributes that composed the primary keys of each table in DW1 and DW10 datasets. We did not create any additional indices.

The workload was composed of SSB's queries, which are organized in four groups of queries Q1, Q2, Q3 and Q4 and have increasing complexity [18]. Each group of query determines an intrinsic number of joins and filters, as well as groupings and sorting. Fig. 12 illustrates each query group template. These templates are described in terms of the operations that are computed to execute their queries in Table 2. Since the queries have filters in the WHERE clause, they enable *slice-and-dice* operations. The *drill-down* and *roll-up* operations were evaluated following the SSB's queries Q3.1, Q3.2, Q3.3 and Q3.4. Executing them progressively determines a *drill-down* operation, while the inverse execution consists of a *roll-up* operation. These queries are shown in Fig. 13, and the attributes used for *drill-down* and *roll-up* operations are highlighted in bold.

Other datasets were created as *vertically fragmented views* and *materialized views*, similarly to those of Section 2.2. Their descriptions are provided in the next sections. In addition to low cardinality attributes, all queries involve

at least one high cardinality attribute. For instance, *lo\_revenue* attribute has a cardinality of 3,345,588 in the DW1 dataset, and a cardinality of 5,841,774 in the DW10 dataset. These attributes were used aiming at assessing our tool when dealing with high cardinality.

<p><b>Q1</b>  SELECT SUM(lo_extendedprice*lo_discount) AS revenue  FROM Lineorder, Date  WHERE lo_orderdate = d_datekey  AND d_year = [YEAR]  AND lo_discount BETWEEN [DISCOUNT] - 1  AND [DISCOUNT] + 1  AND lo_quantity &lt; [QUANTITY];</p>	<p><b>Q2</b>  SELECT SUM(lo_revenue), d_year, p_brand1  FROM Lineorder, Date, Part, Supplier  WHERE lo_orderdate = d_datekey  AND lo_partkey = p_partkey  AND lo_suppkey = s_suppkey  AND p_category = 'MFGR#12'  AND s_region = 'AMERICA'  GROUP BY d_year, p_brand1  ORDER BY d_year, p_brand1;</p>
<p><b>Q3</b>  SELECT c_nation, s_nation, d_year,  SUM(lo_revenue) AS revenue  FROM Customer, Lineorder, Supplier, Date  WHERE lo_custkey = c_custkey  AND lo_suppkey = s_suppkey  AND lo_orderdate = d_datekey  AND c_region = 'ASIA'  AND s_region = 'ASIA'  AND d_year &gt;= 1992 AND d_year &lt;= 1997  GROUP BY c_nation, s_nation, d_year  ORDER BY d_year asc, revenue DESC;</p>	<p><b>Q4</b>  SELECT d_year, c_nation,  SUM(lo_revenue - lo_supplycost) AS profit  FROM Date, Customer, Supplier, Part, Lineorder  WHERE lo_custkey = c_custkey  AND lo_suppkey = s_suppkey  AND lo_partkey = p_partkey  AND lo_orderdate = d_datekey  AND c_region = 'AMERICA'  AND s_region = 'AMERICA'  AND (p_mfgr = 'MFGR#1' OR p_mfgr = 'MFGR#2')  GROUP BY d_year, c_nation  ORDER BY d_year, c_nation;</p>

**Fig. 12:** The templates for the SSB's queries [18].

**Table 2:** Description of the queries templates in Fig. 12

Query Group	Joins	Filters	Aggregation?	Sorting?	Attributes to be indexed
Q1	1	3	No	No	4
Q2	3	2	Yes	Yes	5
Q3	3	3	Yes	Yes	7
Q4	4	3	Yes	Yes	7

<p><b>Q3.1</b>  SELECT c_nation, s_nation, d_year, sum(lo_revenue) AS revenue  FROM customer, lineorder, supplier, date  WHERE lo_custkey = c_custkey  AND lo_suppkey = s_suppkey  AND lo_orderdate = d_datekey  AND c_region = 'ASIA' AND s_region = 'ASIA'  AND d_year &gt;= 1992 AND d_year &lt;= 1997  GROUP BY c_nation, s_nation, d_year  ORDER BY d_year ASC, revenue DESC;</p>	<p><b>Q3.2</b>  SELECT c_city, s_city, d_year, sum(lo_revenue) AS revenue  FROM customer, lineorder, supplier, date  WHERE lo_custkey = c_custkey  AND lo_suppkey = s_suppkey  AND lo_orderdate = d_datekey  AND c_nation = 'JAPAN' AND s_nation = 'JAPAN'  AND d_year &gt;= 1992 AND d_year &lt;= 1997  GROUP BY c_city, s_city, d_year  ORDER BY d_year ASC, revenue DESC;</p>
<p><b>Q3.3</b>  SELECT c_city, s_city, d_year, sum(lo_revenue) AS revenue  FROM customer, lineorder, supplier, date  WHERE lo_custkey = c_custkey  AND lo_suppkey = s_suppkey  AND lo_orderdate = d_datekey  AND (c_city = 'JAPAN 1' OR c_city = 'JAPAN 5')  AND (s_city = 'JAPAN 1' OR s_city = 'JAPAN 5')  AND d_year &gt;= 1992 and d_year &lt;= 1997  GROUP BY c_city, s_city, d_year  ORDER BY d_year ASC, revenue DESC;</p>	<p><b>Q3.4</b>  SELECT c_city, s_city, d_year, sum(lo_revenue) AS revenue  FROM customer, lineorder, supplier, date  WHERE lo_custkey = c_custkey  AND lo_suppkey = s_suppkey  AND lo_orderdate = d_datekey  AND (c_city = 'JAPAN 1' OR c_city = 'JAPAN 5')  AND (s_city = 'JAPAN 1' OR s_city = 'JAPAN 5')  AND d_yearmonth = 'Dec1997'  GROUP BY c_city, s_city, d_year  ORDER BY d_year asc, revenue DESC;</p>

**Fig. 13:** Adapted queries to evaluate drill-down and roll-up operations

The hardware and software platforms used are described as follows.

- Platform P1 was a computer with an Intel® Core™ 2 Duo processor with frequency of 2.80GHz, 320 GB SATA hard drive with 7200 RPM, and 3 GB of main memory. The operating system was CentOS 5.4 with Kernel Version 2.6.18-164.el5, and the following softwares were installed: FastBit 1.2.2, PostgreSQL 8.4, JDK 1.6.0\_21 and Apache Tomcat 6.0.29; and
- Platform P2 comprised a computer with an Intel® Core™ i5 processor with frequency of 2.66GHz, 640 GB SATA hard drive with 7200 RPM, 4 GB of main memory, Ubuntu 10.10 with Kernel 2.6.35-27, FastBit 1.2.4, PostgreSQL 9.0, JDK 1.6.0\_24 and Apache Tomcat 7.0.14.

Two distinct platforms were utilized due to the costly operations involved. Both P1 and P2 platforms had Open Ajax Toolkit 2.8, Mondrian Schema Workbench 3.2.0 and Mondrian OLAP Server 3.2.1.13885 installed. Finally, all bitmap join indices were built with WAH compression algorithm, equality encoding and no binning. These features are enabled by the FastBit by default to improve Bitmap indices over high cardinality attributes [14].

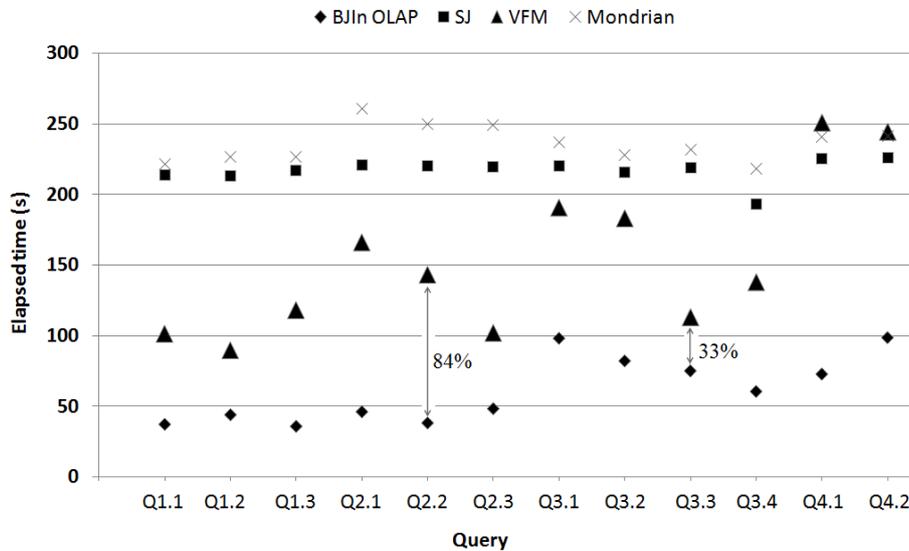
#### 4.2 Comparing the BJIn OLAP Tool to vertically fragmented views

These experiments were conducted in platform P1 and considered the following configurations to execute queries:

- *SJ* used the DBMS to compute the star-join on the DW1 dataset;
- *VFM* used the DBMS to avoid joins by accessing a specific vertically fragmented view that was previously built over the DW1 dataset;
- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices that were previously built over the DW1 dataset; and
- *Mondrian OLAP Server* to access the DW1 dataset using MDX.

Note that the VFM configuration demanded the creation of one vertically fragmented view for each SSB query, similarly to that view of Section 2.2. We performed all tests locally to avoid network latency. All SSB's queries were issued, and the system cache was flushed after the execution of each query. We gathered the elapsed time in seconds to process each query. The results were reported in Fig. 14.

Clearly, the BJIn OLAP Tool outperformed all the other configurations, corroborating the use of the bitmap join index to process OLAP queries. On the other hand, the Mondrian configuration was the one that mostly impaired the query processing performance. In fact, OLAP servers often access the star schema maintained by the DBMS in order to perform the queries, mapping MDX to SQL queries. Therefore, as the Mondrian configuration accessed DW1 just as the SJ configuration did, it was already expected that they would obtain similar results. Furthermore, there was an overhead that differed Mondrian and SJ configurations, since only the former needed to prepare Java Server Pages and render cross tables to show to the user. Both the SJ and the Mondrian configurations provided unacceptable query response times.



**Fig. 14:** Elapsed time obtained by each configuration to process SSB's queries

Moreover, the VFM configuration overcame the SJ configuration since the former avoids joins. The exceptions were queries Q4.1 and Q4.2, where the VFM configuration performed a sequential scan on the text attribute *p\_mfgr*

introducing an overhead. However, the improvement provided by the VFM configuration was smaller than the improvement achieved by our tool to process queries. Actually, the time reduction imposed by the BJIIn OLAP Tool over vertically fragmented views ranged from 33% in Q3.3 up to 84% in Q2.2. The time reduction is a percentage that determines how much more efficient one configuration was than other configuration. Note that, as every query of the workload had restrictions in the WHERE clause, the results corroborated the use of the bitmap join index in OLAP tools to improve the performance of the *slice-and-dice* operation.

The attribute’s cardinality is a very important issue whenever dealing with bitmap indices, since it determines the quantity of bit-vectors built for the corresponding attribute. Fig. 15 illustrates, for the indices built for each query, the quantity of bit-vectors available, i.e. the sum of the cardinalities of all indexed attributes. Only the indices of group Q1 have less than 3 million bit-vectors. According to our assessments, every query execution accessed more than 99% of the available bit-vectors. Therefore, the results revealed that the BJIIn OLAP Tool efficiently performed queries even for very high cardinalities and accessing a huge number of bit-vectors.

The construction of the bitmap join indices spent 1,896 seconds, while the vertically fragmented views accessed by the DBMS spent 6,225 seconds to be built. Regarding storage, Fig. 16 shows individual requirements for both vertically fragmented views (VFM) and bitmap join indices that were built to process each SSB query. Naturally, as more bit-vectors need to be built (Fig. 15), more storage space is required (Fig. 16). As a result, VFMs and bitmap join indices that were built for group Q1 required less storage space than other groups. Also, for group Q1, the bitmap join indices occupied less space than vertically fragmented views.

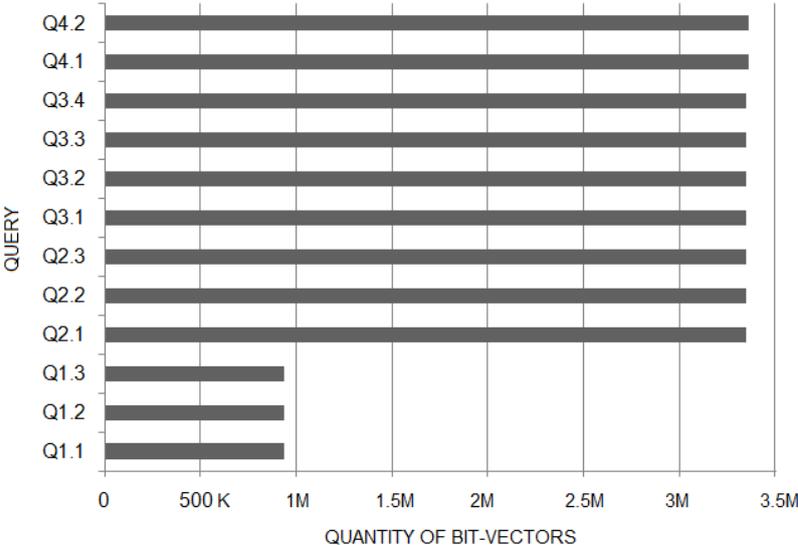


Fig. 15: Quantity of bit-vectors available for the index of each query

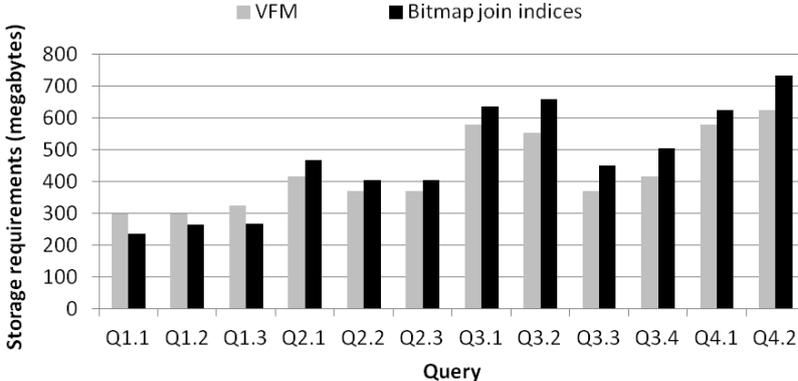


Fig. 16: Storage requirements for both the vertically fragmented views and the bitmap join indices

On the other hand, considering groups Q2, Q3 and Q4, indices required more storage space than views. Although the cardinalities of the attributes in groups Q2, Q3 and Q4 were similar (Fig. 15), the storage requirements varied according to each query as shown in Fig. 16. This difference is due to the existence of attributes of distinct data

types (and sizes in bytes) in each query. For instance, although queries Q2.1 and Q4.2 had bitmap join indices with similar cardinalities built (Fig. 15), their indices occupied distinct amount of storage space (Fig. 16). Finally, the DW1 dataset occupied 838 MB, the created vertical fragmented views occupied a sum of 5,193 MB, and the bitmap join indices occupied a sum of 5,652 MB. Compared to vertically fragmented views, bitmap join indices required 8.8% more disk space. However, these indices have reasonably improved the query processing performance and spent less time to be built.

### 4.3 Drill-down and roll-up operations

In this test we evaluated *drill-down* and *roll-up* operations using platform P1 and the following test configurations:

- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices built over the DW1 dataset, comprising the attributes *d\_year*, *d\_yearmonth*, *s\_region*, *s\_nation*, *s\_city*, *c\_region*, *c\_nation*, *c\_city* and *lo\_revenue*; and
- *Mondrian OLAP Server* to access the DW1 dataset using MDX.

We executed the queries shown in Fig. 13 consecutively without flushing the system cache between each query. This strategy allows the cache to be used and therefore to rapidly fetch partial results of the query. We performed both the *drill-down* and *roll-up* operations five times, gathered the elapsed time of each specific query and later calculated the average. Also, we calculated the total elapsed time of each OLAP operation and the time reduction provided by the BJIn OLAP Tool over Mondrian.

The results were reported in Table 3, and revealed that the first query, i.e. the query Q3.1, was the costly query of the *drill-down* operation. This fact confirmed the importance of the cache whenever performing this OLAP operation, in order to rapidly fetch partial query results, and then provide a shorter elapsed time to process the subsequent queries. An important result derived from our experiments is that the BJIn OLAP Tool greatly outperformed the Mondrian configuration to execute the first query of both *drill-down* and *roll-up* operations, i.e. Q3.1 and Q3.4, respectively.

The experiments had also shown that the BJIn OLAP Tool drastically decreased the query response time to process *drill-down* and *roll-up* operations. Actually, our tool provided a time reduction of at least 45% over Mondrian. This fact corroborated the use of the bitmap join index in OLAP tools in order to improve the query processing performance of *drill-down* and *roll-up* operations.

**Table 3:** Drill-down and roll-up operations performed by Mondrian and the BJIn OLAP Tool

Query	Drill-down (s): Q3.1 down to Q3.4		Roll-up (s): Q3.4 up to Q3.1	
	Mondrian	BJIn OLAP Tool	Mondrian	BJIn OLAP Tool
Q3.1	226.553	73.835	4.356	50.067
Q3.2	3.738	52.651	3.390	5.482
Q3.3	0.239	0.648	2.875	29.708
Q3.4	2.866	0.411	231.313	45.403
Total	233.396	127.545	241.934	130.660
<b>Time Reduction (%)</b>	45.35%		45.99%	

### 4.4 Increasing Data Volume and Accessing Materialized Views

In order to assess our BJIn OLAP Tool for its efficiency and scalability, we performed experiments with a greater data volume (DW10) than those used in sections 4.2 and 4.3 (DW1), and the same platform P1. Besides, in this section we state four new configurations as follows:

- *DBMS+MV* was the DBMS avoiding joins by accessing specific materialized views that were built to process each one of the SSB's queries over the DW10 dataset (similarly to the view shown in Fig. 5a);
- *Mondrian+MV* was the Mondrian OLAP Server accessing the previously cited materialized views using MDX and Aggregate Tables;
- *FastBit* used the FastBit to avoid joins by accessing bitmap join indices that were previously built over the cited materialized views (similarly to Fig. 5b); and
- *BJIn OLAP Tool* to avoid joins by accessing the previously mentioned bitmap join indices.

Although the DW10 had a greater volume, materialized views drastically reduced the quantity of rows. We performed all tests locally to avoid network latency. This test also compared the performance of the OLAP tool and its query engine, aiming to estimate the overhead, i.e. the difference between the time spent by the OLAP tool and the time spent by the query engine. While the query engine of Mondrian is the DBMS, the query engine of the BJIn OLAP Tool is the FastBit. All queries of the SSB were issued, and the system cache was flushed after the execution of each query. We gathered the elapsed time in seconds to process each query. The results were reported in Fig. 17.

The results revealed that the BJIn OLAP Tool outperformed the Mondrian configuration in every query. Concerning the engines, the FastBit outperformed the DBMS in most queries, except for group Q1, which has a very

low volume (less than 100 rows). Actually, the time reduction imposed by the BJIIn OLAP Tool over the Mondrian configuration ranged from 71% in Q1.2 up to 97% in Q3.4. Therefore, our tool demonstrated to be feasible when indexing materialized views and processing queries using these indices.

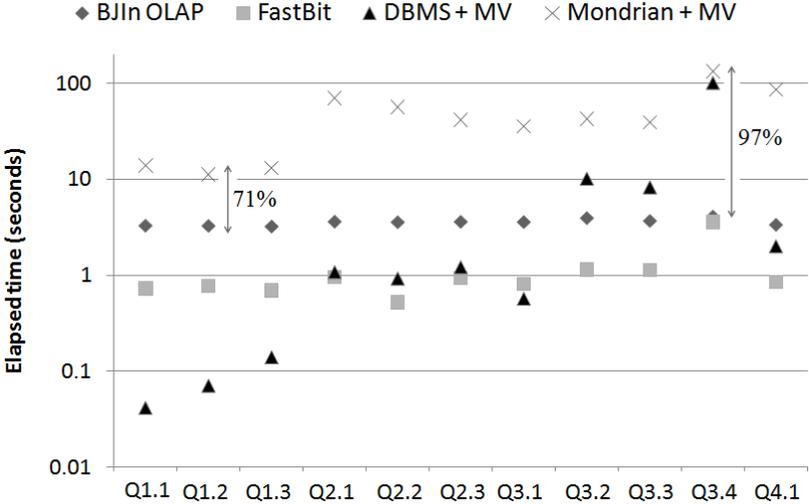


Fig. 17: Elapsed time to process SSB’s queries on DW10 dataset, in logarithmic scale (base 10)

Another interesting result showed that the BJIIn OLAP Tool was capable of displaying the query results much more rapidly than the Mondrian configuration. Fig. 18 shows how many seconds of overhead each OLAP tool added to the elapsed time spent by the query engine to process each query. In other words, it represents the difference between the OLAP tool elapsed time and the engine elapsed time, derived from Fig. 17. Although the DBMS provides the query answer quickly, there is an overhead that severely impaired the Mondrian configuration performance to cache the data cube, translate MDX and render the results. This overhead ranged from 11 seconds (Q1.2) to 83 seconds (Q4.1). This severe overhead was not observed on the BJIIn OLAP Tool, which introduced only a few seconds to the FastBit elapsed time, i.e. at most 3 seconds (Q2.2).

Regarding storage, the DW10 dataset occupied 10,540 MB, all bitmap join indices required 51 MB and all materialized views required 45.6 MB. The construction of the materialized views by the DBMS spent 174,416 seconds, while the indices spent 163 seconds to be built over these views. Although the indices added approximately 12% of storage requirements to materialized views, they greatly improved the query processing performance over the DW10 dataset. Also, the time to build the indices over the materialized views added only 0.00094% to the elapsed time to build these views.

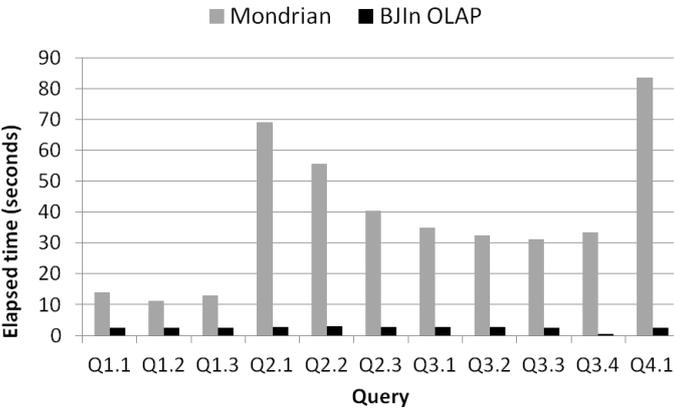


Fig. 18: The overhead tool added to the elapsed time spent by the query engine

**4.5 Rendering interface components on the client side to present the query results**

In this section we evaluate the time spent by OLAP tools to render the interface components and present the query results to the user. Differently from the previous and remaining sections, this test assesses the performance on the client side. This evaluation was motivated by the fact that OLAP tools added significant overheads to the query

processing elapsed times of their corresponding query engines, as discussed in Section 4.5. We measured how much time Mondrian and the BJIIn OLAP Tool spent to load the entire Java Server Page, starting at the moment when the first byte transferred from the server became available, and finishing when the page became completely loaded on the client’s Internet browser. We decided to use the queries of group Q3 (Fig. 13) because they were the most costly according to the results of sections 4.1 to 4.4. Table 4 describes how many rows and columns composed the cross table to display the results of each query regarding the DW1 and DW10 datasets. Note that these datasets have different data volumes as they were generated according to SSB’s scale factors 1 and 10, respectively. As a result, the quantity of rows for the DW1 and the DW10 datasets are not the same in Table 4. All tests were executed in platform P2 due to the complexity of the involved operations. We utilized the Mozilla Firefox 3.6.15 as Internet browser and gathered the elapsed time to load the pages with the utility FireBug 1.7.3.

**Table 4:** Rows and columns that compose the cross table showing the queries’ results

	Q3.1	Q3.2	Q3.3	Q3.4
<b>Columns</b>	c_nation, s_nation, d_year, sum(lo_revenue)	c_city, s_city, d_year, sum(lo_revenue)	c_city, s_city, d_year, sum(lo_revenue)	c_city, s_city, d_year, sum(lo_revenue)
<b>Rows DW1</b>	150	596	24	3
<b>Rows DW10</b>	150	600	24	4

Firstly, we issued five times the queries of group Q3 over the DW1 dataset, gathered the elapsed time to load the page, and then calculated the average. The system cache was flushed between each query execution. We evaluated the following configurations:

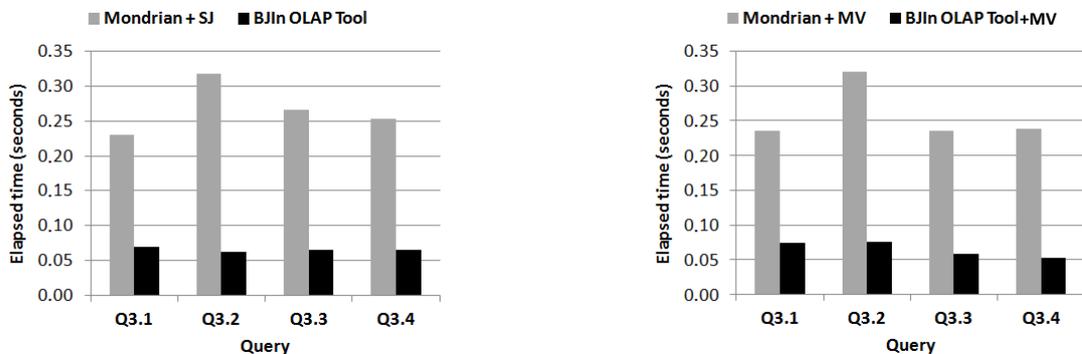
- *Mondrian+SJ* accessed the DW1 dataset using MDX; and
- *BJIIn OLAP Tool* avoided joins by accessing bitmap join indices built for each query of group Q3.

The results are shown in Fig. 19a. They revealed that the BJIIn OLAP Tool was three to five times more efficient than Mondrian to present the query results for the client on the Internet browser, considering Q3.1 and Q3.2, respectively. In addition, the BJIIn OLAP Tool spent the shorter time to render the larger cross table, i.e., for Q3.2.

Secondly, we rerun the experiment over the DW10 dataset and evaluated the following configurations:

- *Mondrian+MV* accessed specific materialized views that were built to process each one of the four Q3 queries; and
- *BJIIn OLAP Tool+MV* avoided joins by accessing bitmap join indices built over the mentioned materialized views.

The results are shown in Fig. 19b and revealed that rendering the results of queries over a more voluminous DW did not impair the performance of both the BJIIn OLAP Tool and Mondrian, if compared to the results of Fig. 19a. Although the query results and the number of rows of the cross table slightly differed (due to the different scale factors chosen for data generation, 1 and 10), there was no significant modifications on performance. Furthermore, the BJIIn OLAP Tool outperformed Mondrian in all queries. Since the results of queries processed by the BJIIn OLAP Tool are written in a CSV file (Fig. 9) and sent through JSON to the client, if a given query is processed by more voluminous indices (e.g. *BJIIn OLAP Tool* configuration) or less voluminous indices (e.g. *BJIIn OLAP Tool+MV* configuration), the performances to load the results on the query browser were similar and independent of the index data volume.



(a) tests executed over the DW1 dataset

(b) tests executed over the DW10 dataset

**Fig. 19:** Elapsed time to load the pages with the answers of the queries in group Q3.

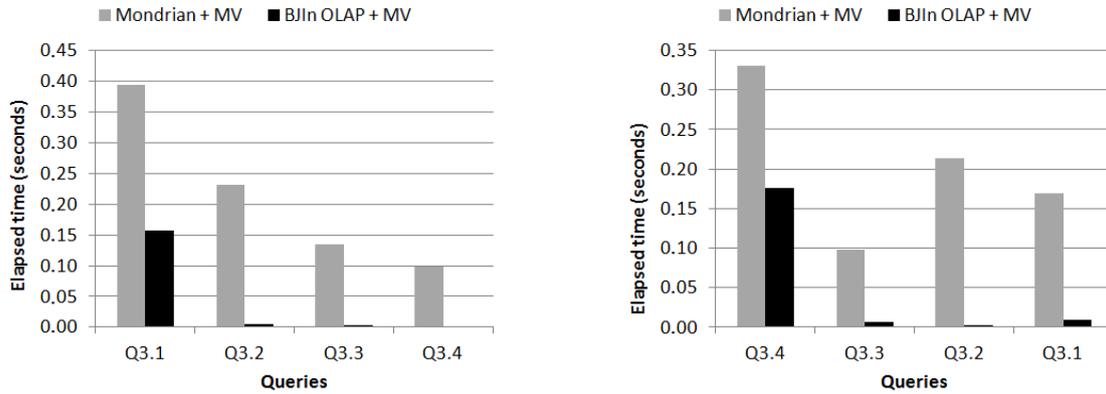
Finally, we issued five times the *drill-down* and the *roll-up* operations of query group Q3 over the DW10 dataset, gathered the elapsed time of each query and calculated the average of these five executions. The system cache was

not flushed between each query execution, aiming at fetching previously computed results in the cache. We evaluated the following configurations:

- *Mondrian+MV* avoided joins by accessing a specific materialized view that was built to process all queries of the Q3 query group (i.e. Aggregate Table); and
- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices that were built over the previously cited materialized view.

The results for the *drill-down* operation are shown in Fig. 20a and revealed that the BJIn OLAP Tool outperformed Mondrian in all queries. Also, our tool has proven to efficiently reuse cache and JSON to render the interface, since it drastically reduced the elapsed time to display the results of the subsequent queries that followed Q3.1. BJIn OLAP Tool also provided a maximum time reduction of 97% when executing Q3.2 after Q3.1, and a minimum time reduction of 8% when executing Q3.4 after Q3.3.

The results for the *roll-up* operation are shown in Fig. 20b and revealed that the BJIn OLAP Tool outperformed Mondrian in all queries. Differently from the *drill-down* operation, the *roll-up* operation did not indicate decreasing response times when executing consecutive queries, for both the BJIn OLAP Tool and Mondrian. Particularly, there was an increase when executing the query Q3.1 after the query Q3.2 with our tool. This fact can be explained by the execution of a data aggregation that reduced 600 rows (Q3.2) to 150 rows (Q3.1) as shown in Table 4, causing an overhead. Even though, the response times to present the queries' results were not greater than 0.009s. Again, the BJIn OLAP Tool drastically reduced the elapsed time to display the results of the subsequent queries that followed Q3.4, due to the use of cache and JSON.



(a) interface test on the drill-down operation

(b) interface test on the roll-up operation

**Fig. 20:** Elapsed time to load the pages with the answers of *drill-down* and *roll-up* operations.

In short, the results presented in this section corroborated the reasonable performance of the interface components adopted by the BJIn OLAP Tool, and provided by the Open Ajax Toolkit Framework. Furthermore, these results were achieved flushing the cache when issuing individual queries (Fig. 19), or maintaining the cache to reuse previously fetched results and then benefit *drill-down* and *roll-up* operations (Fig. 20).

#### 4.6 Memory usage

In this section we present a test that measured and compared the amount of main memory utilized by each OLAP tool. The web applications of Mondrian and of the BJIn OLAP Tool consumed the Java Virtual Machine heap that was measured using the NetBeans Profiler. This software is widely used by developers since is integrated to the NetBeans IDE and is free of charge. Also, the DBMS and the FastBit utilized the main memory managed by the operating system, which was measured using the *ps\_mem.py* library. We summed the amount of memory of the OLAP tools and their corresponding query engines. In this section, all tests were run in platform P2 due to the complexity of the involved operations.

Firstly, we issued the Q3 *roll-up* operation (Fig. 13) over the DW1 dataset and evaluated the following configurations:

- *Mondrian* was the Mondrian OLAP Server accessing the DW1 with the DBMS as query engine; and
- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices with the FastBit as query engine.

Fig. 21 shows the results. The BJIn OLAP Tool had peaks of memory usage whenever one of the four queries was submitted to FastBit. These peaks indicated much more memory usage than Mondrian. However, the BJIn

OLAP Tool spent a shorter time to provide the query answer. We concluded that, for the *roll-up* operation, our tool required more memory in a shorter period, while Mondrian required less and increasing memory for a longer period.

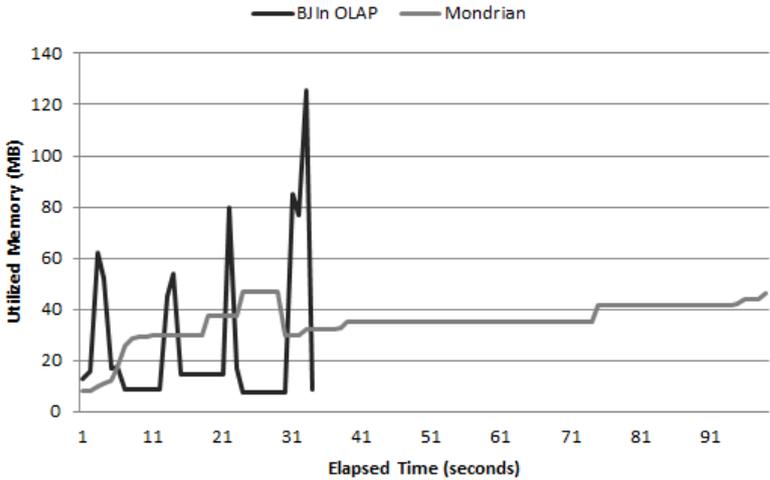


Fig. 21: Memory usage for the roll-up operation over DW1 dataset

Secondly, we issued the query Q3.4 (Fig. 13) over the DW10 dataset and evaluated the following configurations:

- *Mondrian* was the Mondrian OLAP Server accessing the materialized view using MDX and an Aggregate Table, using the DBMS as query engine;
- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices that were built over the cited materialized view, using the FastBit as query engine.

The results are shown in Fig. 22. Although the BJIn OLAP Tool consumed more memory than Mondrian during the initial 6 seconds of execution, it provided a much shorter elapsed time than the latter. Also, Mondrian drastically increased its memory consumption after 6 seconds of execution. The FastBit introduced only one peak of memory consumption (totalizing 55MB) that lasted around 1 second. Again, our OLAP tool had a feasible memory usage, even considering the most costly query (according to Fig. 17).

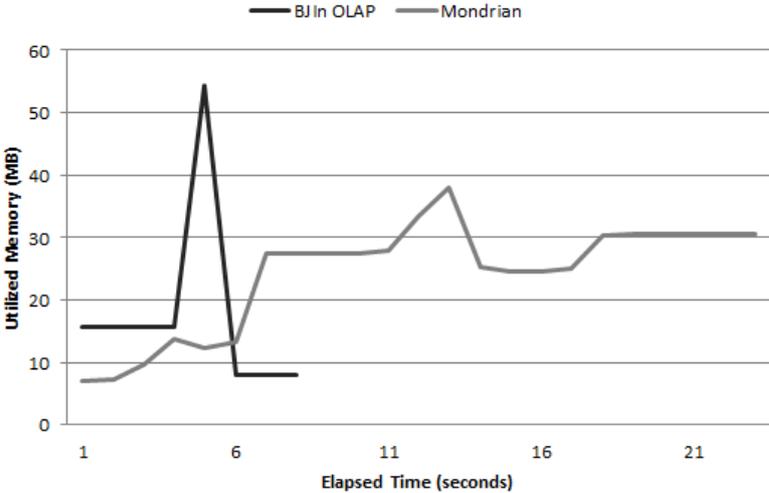


Fig. 22: Memory usage for the query Q3.4 issued over the DW10 dataset

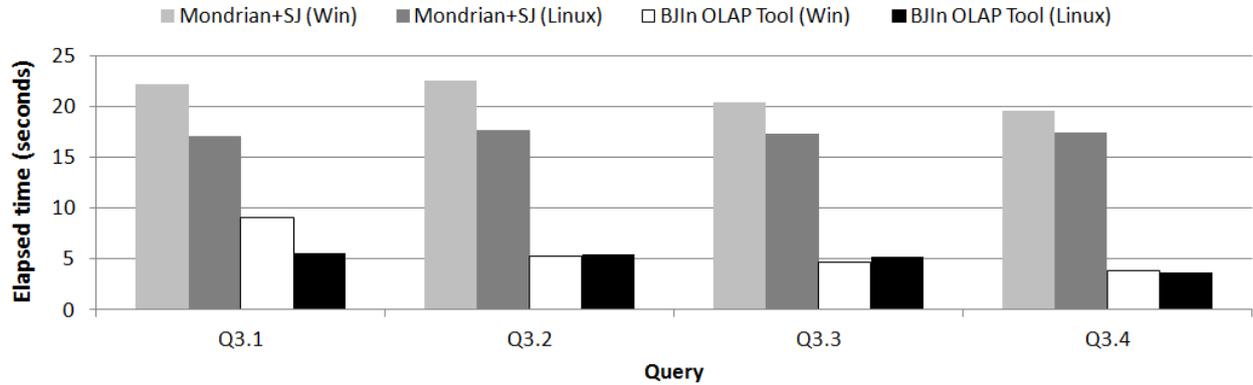
**4.7 Portability and the query processing performance**

In this section we assess the query processing performance of both Mondrian and the BJIn OLAP Tool in two different operating systems aiming at testing the portability. In addition to Linux Ubuntu, we installed Microsoft Windows 7 SP1 Professional 64 bits in the platform P2. Firstly, we issued five times the queries of group Q3 (Fig. 13) over the DW1 dataset, gathered the elapsed time of each query and calculated the average of these five

executions. The system cache was flushed between each query execution. We evaluated the following configurations in both operating systems (namely Win and Linux):

- *Mondrian+SJ* accessed the DW1 dataset using MDX; and
- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices built over the DW1 dataset.

The results are shown in Fig. 23. Clearly, the BJIn OLAP Tool outperformed Mondrian in both operating systems. Comparing Mondrian to itself, it was notably more efficient in one of the operating systems. On the other hand, our tool achieved similar results in most queries, independently from the operating system.

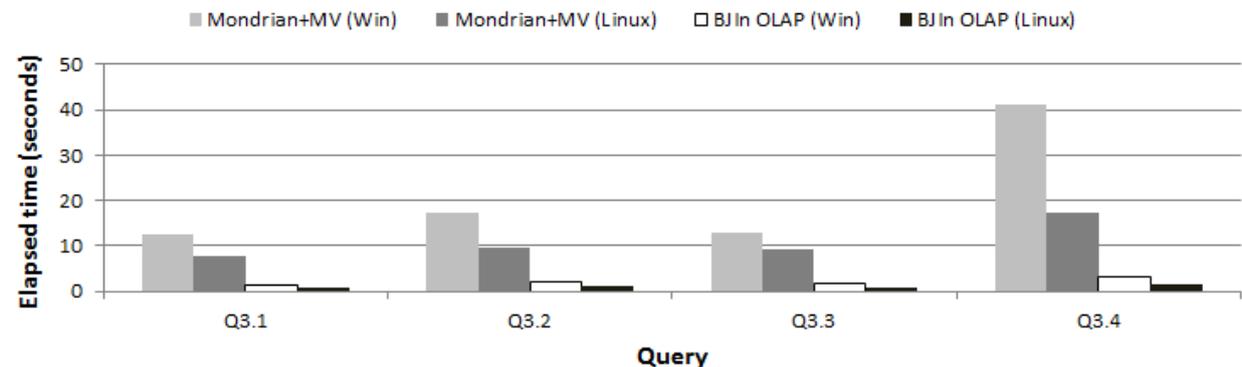


**Fig. 23:** Portability and the query processing performance for the DW1 dataset

Secondly, we repeated the previously described procedures on DW10 dataset to evaluate the following configurations in both operating system (namely Win and Linux):

- *Mondrian+MV* avoided joins by accessing a specific materialized views that was built to process all queries of the Q3 query group (i.e. an Aggregate Table); and
- *BJIn OLAP Tool* avoided joins by accessing bitmap join indices that were built over the previously cited materialized view.

The results are shown in Fig. 24. Again, the BJIn OLAP Tool outperformed Mondrian in both operating systems. Comparing Mondrian to itself, it was notably more efficient in one of the operating systems. Both Mondrian and the BJIn OLAP Tool were more efficient in the Linux operating system when dealing with materialized views and indices built over materialized views, respectively.



**Fig. 24:** Portability and the query processing performance for the DW10 dataset

#### 4.8 Appending new rows to bitmap join indices

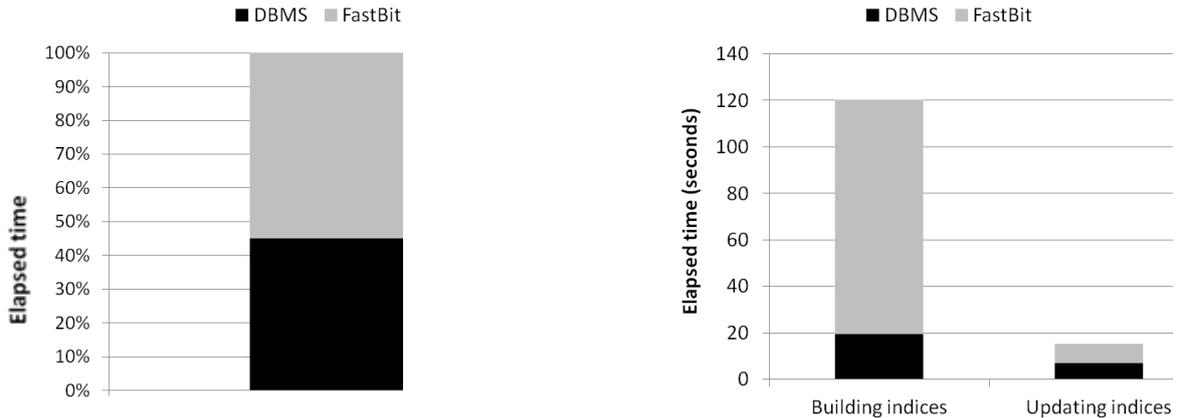
In this section we evaluate the performance of the append operation over bitmap join indices maintained by the BJIn OLAP Tool. We utilized the platform P2, the DW1 dataset, and the most costly bitmap join indices concerning storage requirements, i.e., those indices built to answer the query Q4.2 (see Fig. 16). We also introduced a workload composed of 10% of the original index volume, i.e., 600,000 new rows to be appended to the fact table *Lineorder*. In addition, new values were also introduced in the dimension tables of the DW. As a result, the operation not only appended new bits to existing bit-vectors, but also created new bit-vectors and increased the attributes' cardinalities as shown in **Table 5**. Each new bit-vector created had a minimum of 6 million bits with value 0, since these bits refer to the former existing rows of the index.

**Table 5:** How many new bit-vectors were created for each indexed attribute

Attribute	<i>d_year</i>	<i>s_nation</i>	<i>p_category</i>	<i>lo_revenue</i>	<i>lo_supplycost</i>	<i>s_region</i>	<i>c_region</i>	<i>p_mfgr</i>
New bit-vectors	2	3	5	0	0	0	0	2

The first test assessed the performance of the insertion of new rows in the DW1 dataset. The insertion of the new rows spent 184.03 seconds considering the DW1 dataset with the append trigger. On the other hand, the same insertion took 126.89 seconds without the trigger. Therefore, the execution of the trigger introduced an overhead of 31.05% to the normal insertion in the DW1 dataset. This overhead should be taken into account when using the BJIIn OLAP Tool, since it delays the insertion of new rows into the DW tables. Particularly, the delay is caused mainly due to several queries issued by the trigger before the insertion into the temporary table (line 08 of Algorithm 1).

The second test evaluated the time to process the following tasks related to the append operation of the BJIIn OLAP Tool: dumping the temporary table using the DBMS and executing *ardea* and *ibis* commands in the FastBit to append new values and create new bit-vectors. The whole process took 15.45 seconds. Fig. 25a reveals that dumping the temporary table consumed 45.12% of the time spent for the append operation. On the other hand, the manipulation of the indices consumed 54.88% of this time. Therefore, the tasks performed by the FastBit demonstrated to be more costly than those performed by the DBMS.



(a) tasks developed by the DBMS and the FastBit (b) comparison between building and append operations

**Fig. 25:** Tests regarding the append operation

Finally, the last test compared the building operation to the append operation, according to the tasks performed by the DBMS and the FastBit. We aimed to identify if rebuilding the indices would be more advantageous than appending new rows to them. The results are shown in Fig. 25b. The FastBit execution spent the major parcel of the elapsed time to build the indices, as already identified for the task of appending new rows to indices (Fig. 25a). Also, the building operation took 120.33 seconds, while the append operation took only 15.45 seconds. Therefore, appending new rows to the indices has shown to be more advantageous than rebuilding these indices, since the append operation represented only 12.84% of the time spent by the building operation.

## 5 Related Work

In Table 6 existing software are compared to the BJIIn OLAP Tool. Oracle BI enables the bitmap join index and new rows to be appended to it. However, the use of this index is recommended only when indexing very low cardinality attributes [5]. Conversely, the FastBit provides the support to index attributes with higher cardinalities for our tool, e.g. *lo\_revenue* whose cardinality is 3,345,588. Furthermore, Oracle BI platform is not open source software, does not consist of a web application and therefore was not employed in our performance evaluation.

Regarding the Mondrian OLAP Server [6], although it supports materialized views to improve the query processing performance, it currently does not support the bitmap join index. Moreover, the user must learn and type MDX to issue queries. On the other hand, the BJIIn OLAP Tool uses this index and supports a query language that is syntactically based on SQL. Although this query language does not provide OLAP operations, the user is able to perform them from the BJIIn OLAP Tool graphical user interface. As a result, the BJIIn OLAP Tool does not require knowledge about additional query languages, such as MDX that is commonly adopted by existing OLAP Tools [6][5][4][20]. Besides, Mondrian does not implement any mechanisms to append new rows to materialized views, while the BJIIn OLAP Tool enables new rows to be appended to bitmap join indices.

Important issues regarding the design of the interfaces of Mondrian and the BJIIn OLAP Tool need to be stated. The former employs synchronous requests to transfer data from the server to the client. As a result, the server

retrieves the query answer and sends preprocessed cross tables and charts to the client, which interprets the HTML code and renders the components. On the other hand, the latter employs asynchronous requests through Ajax and processes JSON [26] sent by the server on the client side. In short, our tool processes interface components on the client side, while Mondrian does not. We did not consider existing unofficial Mondrian extensions.

**Table 6:** Comparison among existing software and the BJIIn OLAP Tool

Feature	Oracle BI	Mondrian	FastBit	BJIIn OLAP Tool
Supports the bitmap join index	✓		✓	✓
Supports OLAP operations	✓	✓		✓
Avoids typing complex OS commands	✓	✓		✓
Avoids typing complex DBMS commands		✓		✓
Provides information visualization methods	✓	✓		✓
Is open source software		✓	✓	✓
Is a web application		✓		✓
Implements the append operation	✓		✓	✓
Uses asynchronous requests and JSON				✓

In addition, the BJIIn OLAP Tool interestingly uses the FastBit to be applied over DW and support *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* OLAP operations. Although the FastBit natively supports the bitmap join index, building, appending rows to and dropping this index require many long and complex instructions to be typed by the user, both in the operating system and DBMS command lines. The BJIIn OLAP Tool avoids keyboarding by enabling a graphical interface for the user. Also, our tool operates the FastBit on the server side, while in the client side the results of queries are displayed to the user in cross tables and charts via the web.

The experimental evaluation presented in Section 4 aimed at testing the BJIIn OLAP Tool. We investigated the feasibility of developing an OLAP tool exclusively based on the bitmap join index to process OLAP operations such as *drill-down*, *roll-up*, *slice-and-dice* and *pivoting*, comparing it to existing software. Differently from [27], our goal was not to exhaustively assess the bitmap join index against other join indices. Also, we have not considered specific algorithms to select the bitmap join indices, as already stated by [15].

## 6 Conclusions and Future Work

In this paper, we introduced the BJIIn OLAP Tool to efficiently perform *drill-down*, *roll-up*, *slice-and-dice* and *pivoting* OLAP operations over DW, by employing the bitmap join index. The BJIIn OLAP Tool is Free Software and was implemented and tested through a performance evaluation to assess its efficiency and to corroborate the feasibility of adopting strictly the bitmap join index in an OLAP tool. The performance results reported that our BJIIn OLAP Tool provided a performance gain that ranged from 31% up to 97% if compared to existing solutions regarding the query processing, even for attributes with a very high cardinality. Furthermore, our tool has proven to efficiently process these operations both on the server and client sides, for different volumes of data and also taking into account different operating systems, enforcing its portability. In addition, the BJIIn OLAP Tool provides a reasonable use of the main memory and enables new rows to be appended to bitmap join indices.

Currently, the BJIIn OLAP Tool is being applied to the Web-PIDE Project over DW containing real educational data from Brazilian Government [28][29], to aid decision takers on planning Educational Policies. Moreover, our tool is under a registration process to be formally registered as Free Software, and its internet portal is under development. As future work, we intend to investigate *drill-across* OLAP operations, adapt the tool for Spatial OLAP operations [17] and enable the use on mobile devices [30]. Also, the BJIIn OLAP Tool will be and its Portal will contain proper documentation.

### Acknowledgements

The 1<sup>st</sup> author thanks the support of IFSP Undergraduate Research Grant.

The 2<sup>nd</sup> author thanks Projeto Web-PIDE (Observatório da Educação: CAPES/INEP).

### References

- [1] R. Wrembel, C. Koncilia, *Data Warehouses and OLAP: Concepts, Architectures and Solutions*. IRM Press, 2006.
- [2] L. Xu, L. Zeng, Z. Shi, Q. He, M. Wang, "Research on business intelligence in enterprise computing environment," *IEEE SMC*, pp.3270-3275, 2007.

- [3] W. H. Inmon, *Building the Data Warehouse*. Wiley, 2002.
- [4] M. Golfarelli, "Open source BI platforms: a functional and architectural comparison," *DaWaK*, pp. 287-297, Springer, 2009.
- [5] S. Fogel, C. Johnston, S. Moore, T. Morales, P. Potineri, R. Urbano, L. Ashdown, J. Greenberg, *Oracle 11g database administrator's guide*. 2010.
- [6] M. Casters, R. Bouman, J. Dongen, *Pentaho® Kettle Solutions*. Sybex, 2010.
- [7] S. Chaudhuri, U. Dayal, "An overview of data warehousing and OLAP technology," *SIGMOD Record*, vol. 26, pp. 65-74, 1997.
- [8] M. Golfarelli, D. Maio, S. Rizzi, "Applying vertical fragmentation techniques in logical design of multidimensional databases," *DaWaK*, pp. 11-23, Springer, 2000.
- [9] E. Baikousi, P. Vassiliadis, "View usability and safety for the answering of top-k queries via materialized views," *DOLAP*, pp. 97-104, ACM, New York, 2009.
- [10] L. Bellatreche, K. Y. Woameno, "Dimension table driven approach to referential partition relational data warehouses," *DOLAP*, pp. 9-16. ACM, New York, 2009.
- [11] V. Harinarayan, A. Rajaraman, J. D. Ullman, "Implementing data cubes efficiently," *SIGMOD Record*, vol. 25, pp. 205-216, 1996.
- [12] A. S. Firmino, R. C. Mateus, V. C. Times, L. F. Cabral, T. L. L. Siqueira, R. R. Ciferri, C. D. A. Ciferri, "A Novel Method for Selecting and Materializing Views based on OLAP Signatures and GRASP," *JIDM*, vol. 2, no. 3, pp. 479-494, 2011.
- [13] P. O'Neil, G. Graefe, "Multi-table joins through bitmapped join indices," *SIGMOD Record*, vol. 24, pp. 8-11, 1995.
- [14] K. Stockinger, K. Wu, "Bitmap indices for data warehouses," in *Data Warehouses and OLAP*, IRM Press, 2006, pp. 157-178.
- [15] L. Bellatreche, K. Boukhalfa, "Yet Another Algorithms for Selecting Bitmap Join Indexes" *DaWaK*, pp. 105-116, 2010.
- [16] A. C. Carniel, T. L. L. Siqueira, "An OLAP Tool based on the Bitmap Join Index," *CLEI*, pp. 911-926, 2011.
- [17] T. L. L. Siqueira, C. D. A. Ciferri, V. C. Times, R. R. Ciferri, "The SB-index and the HSB-Index: efficient indices for spatial data warehouses," *Geoinformatica*, vol.16, no. 1, pp. 165-205, 2011.
- [18] P. O'Neil, E. O'Neil, X. Chen, S. Revilak, "The star schema benchmark and augmented fact table indexing," *TPCTC*, pp. 237-252, 2009.
- [19] R. Kimball, M. Ross, *The data warehouse toolkit: the complete guide to dimensional modeling*. John Wiley & Sons, Inc., Chichester, 2002.
- [20] M. Whitehorn, R. Zare, M. Pasumansky, *Fast Track to MDX*. Springer, 2005.
- [21] N. C. Zakas, J. McPeak, J. Fawcett, "What is Ajax?," in *Professional Ajax*, Wiley Publishing Inc., 2006, pp. 1-45.
- [22] C. Y. Chan, "Bitmap Index," in *Encyclopedia of Database Systems*, Springer, 2009, pp. 244-248.
- [23] O. Rübél, A. Shoshani, A. Sim, K. Stockinger, G. Weber, W. M. Zhang, Prabhat, "FastBit: interactively searching massive data," *J. of Physics: Conference Series*, vol. 180, 12053, 2009.
- [24] C. D. A. Ciferri, F. F. Fonseca, "Materialized Views in Data Warehousing Environments," *SCCC*, pp. 3-12, 2001.
- [25] G. Canahuate, "Update Conscious Bitmap Indexes," in *Enhanced Bitmap Indexes for Large Scale Data Management*, The Ohio State University, Dissertation, ch. 7, pp 141-163, 2009.
- [26] D. Crockford, *JavaScript: The Good Parts*. Yahoo Press, 2008.
- [27] A. Datta, D. VanderMeer, K. Ramamritham, "Parallel Star Join+DataIndexes: efficient query processing in data warehouses and OLAP," *IEEE TKDE*, vol. 14, pp. 1299-1316, 2002.
- [28] A. C. Carniel, T. L. L. Siqueira, "The Bitmap Join Index OLAP Tool," *SBB Demos*, pp. 13-18. 2011.

- [29] T. L. L. Siqueira, R. R. Ciferri, M. T. P. Santos, "Projeto, construção e manutenção de data warehouses para auxiliar o planejamento de políticas públicas de educação," *XVI Jornadas de J6venes Investigadores*, AUGM, pp. 1016-1025, 2008.
- [30] A. S. Maniatis, "The Case for Mobile OLAP," *Pervasive Information Management PIM '04*, Mar., 2004.