The Heterogeneity Problem and Middleware Technology: Experiences with and Performance of Database Gateways

Fernando de Ferreira Rezende
Dept. CAE-Research (FT3/EK)
Daimler-Benz AG - Ulm - Germany
fernando.rezende@dbag.ulm.DaimlerBenz.COM

Klaudia Hergula
Dept. CAE-Research (FT3/EK)
Daimler-Benz AG - Ulm - Germany
klaudia.hergula@dbag.ulm.DaimlerBenz.COM

Abstract

In this paper, we present the results that we have obtained by comparing and testing three well-known database middleware solutions. We have analyzed their features related to global catalog and location transparency, transaction management, DML and DDL operations, SQL-dialects mask, referential integrity, security, scalability, supported data sources and platforms, query optimization, and performance. In particular, we have adapted and implemented the AS^{3}AP benchmark to exhaustively evaluate the performance of the products.

1 Introduction

A common scene within most organizations nowadays is the distribution of data along departmental and functional lines. This leads to fragmented data resources and services, and contributes to the emergence of the so-called "*islands of information*". Data are organized and managed by a mix of different data management systems from different vendors and different operating systems that use different network protocols. In essence, the total corporate data resource is made up of multi-vendor database (DB) servers, legacy and current data, and relational and non-relational data sources. Unfortunately, such autonomous data sources have no ability to relate data from these heterogeneous data sources within the organization.

Middleware is a generic term referring to a system layer of software that tries to overcome the heterogeneity problem. The main goal of middleware is to shield both end-users and programmers from differences in the various services and resources used by the applications. Middleware software aims at simplifying user interfaces by providing a uniform and transparent view of those services and resources that diverge because they are provided by multi-vendors, they have been developed in accordance with different protocols, or are used to support distinct applications. DB gateways constitute a class of DB management system (DBMS) middleware. They provide access from a DB application developed using one vendor's DBMS to a DB processed by the DBMS of a different vendor on the same or perhaps a different platform, shielding the application developer from the differences in the multi-vendor's products.

At Daimler-Benz Research and Technology, we are developing a project called MENTAS (*Motor Development Assistant*) where this heterogeneity problem comes well into the surface. The main goal of MENTAS is the realization of an interconnected, engineer-oriented development environment for a faster conception and comparative analysis of motors. In order to reach this goal, an automatic access to multi-vendor DBs and simulation tools must be provided. Thus, the access to heterogeneous DBs, which are spread along a number of servers of Mercedes-Benz's intranet, shall be coped with by a DB gateway in MENTAS.

There are a lot of DB gateway products commercially available in the market, and each of them promises in its particular way "almost everything". However, many of the promises are merely marketing strategies. Further, as everyone knows, what you hear from a vendor before buying a product is much different from that after you have bought it. In order not to live this situation as well as to become a real feeling about the products' capabilities, we have decided to analyze and critically evaluate some of the most well-known DB gateways available in the market

Proceedings of the 24th VLDB Conference
New York, USA, 1998

today. In this paper we present the results we have obtained in our analysis. We compare the DB gateways with each other and show which the best is, in which particular situation, and under which conditions. We point out the fulfilled expectations at the same time we discuss where they could have done better and what should be considered in next releases of such products. This paper is organized as follows. In Sect. 2, we give a brief overview about our project MENTAS. Sect. 3 introduces the middleware technology and some well-known solutions available today in the market. The comparative analysis between the products is started in Sect. 4. Sect. 5, in turn, presents the performance evaluation. Finally, Sect. 6 concludes the paper and summarizes our experiences with and the performance of DB gateways.

2 Motor Development Assistant

The Motor Development Assistant, or just MENTAS for short, is an innovation project at Daimler-Benz AG Research and Technology in Germany. It has been started in 1997 and is being developed for the mechanical engineers of the Motor Development of Mercedes-Benz with the following motivation in mind: to increase both the capability to react to market changes as well as the potential to innovate, to reduce the development time and consequently the costs, and to parallelize the development work.

The today's motor development environment at Mercedes-Benz is characterized by isolated, multi-vendor software systems and DBs. These systems are, in their majority, calculation and simulation tools and CAD systems which cannot "understand" or communicate with each other. In addition, the DB systems have unfortunately no ability to relate data from heterogeneous data sources. Essentially, each department constitutes an *information island* with its own tools and sources of data. Lastly, the communication and teamwork between such departments is realized in a very informal way, usually via phone calls or by filling up job orders.

MENTAS aims at realizing an interconnected, engineer-oriented development environment for a faster conception and comparative analysis of motors. In order to reach this goal, an automatic, integrated access to heterogeneous DBs and simulation tools must be provided. In this paper, we give emphasis to the DB side of MENTAS and leave the problem of integrating individual simulation tools to another opportunity.

Fig. 1 illustrates how the interconnection of heterogeneous DBs in MENTAS works. As said, each department has its own sources of data. Important to notice here is that the local autonomy of the departments' data sources must be maintained. In practice, these data sources vary *mainly* among different versions of ORACLE DBs and different versions of DB2 DBs.[1] We have analyzed the data models of each such DBs in order to identify the crucial data for MENTAS, to recognize semantic differences, ambiguities, commonalities, synonyms, homonyms, and all the like. Once done that, we have brought the heterogeneous schemas into a global, virtual one, which contains just the data relevant for MENTAS. At this time, we have also analyzed aspects related to the distribution of the data, communication, and consistency. Finally, we apply then a DB middleware to bridge the diverse ontologies and hence to cope with these heterogeneous schemas. By this means, MENTAS can formulate requests as though all data reside in a single local DB when, in fact, most of the data are distributed over heterogeneous remote data sources.

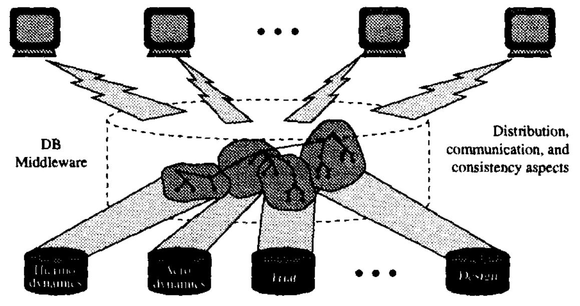

Fig. 1: Integration of heterogeneous schemas.

As suggested until now, a DB middleware solution can bring a lot of benefits. Currently, there are many DB middleware solutions commercially available in the market and they promise in their own way almost everything. Worse, the papers and material available about such products have more an advertising than informative character, lack on technical details of realization and utilization, and were definitively written by marketing people. Due to this and to the fact that this global schema is a potential bottleneck in MENTAS, we have decided to critically analyze, evaluate, and compare the most popular DB middleware solutions in the market today. Our intention was to find out the most appropriate one for the application in MENTAS. In the following, we shortly introduce them and thereafter we present the results of our evaluation.

3 Database Middleware Technology

Whenever an application wants to access DBs managed by multi-vendor DBMSs, code must be written to

[1] Due to its nature, this paper refers to numerous software and hardware products by their trade names. In most cases, if not all, these names are claimed as trademarks or registered trademarks by their respective companies. It is not our intention to use any of these names generically. The reader is therefore cautioned to investigate all claimed trademark rights before using any of these names other than to refer to the product described.

establish separate connections to each, local or remote, data source. The major problem is that the application must be then tailored to the DBMS in question and the requests and queries must be issued using the native interfaces of these data sources. The replacement or insertion of a new source of data requires changes in the applications. DB middleware shields the application developer against these problems by hiding the heterogeneity of network protocols, data sources, and interfaces. It provides one single SQL API and manages all differences in the SQL dialects.

There are mainly two categories of DB middleware products currently available: middleware software and DB gateway. Whereas the middleware software can establish connection to several data sources at once, a DB gateway establishes a point-to-point connection to one remote DB. The gateway approach is primarily used to expand the functionality of one DBMS to another. Notwithstanding, both solutions can be employed to solve the heterogeneity problem. In this paper, otherwise explicitly noted, we use both terms interchangeably with the same meaning.

At the time we have decided to employ the middleware technology to bridge the DB heterogeneity in MENTAS, we have looked up the products currently available in the market. The products we have considered for testing are: *DJ*, *MIRACLE*, and *EDI/S*.[2]

4 Comparative Analysis

4.1 Global Catalog

The global catalog stores the information that the middleware software uses in order to access the remote DBs. With this information, the middleware can find out which node the DB is located in, which type of DB it is, which tables are available, and additionally which its access rights to the DB are. All three products encapsulate the heterogeneity of the DB schemas by using the concept of *synonyms*. The way the synonyms are created and managed by the products is somewhat different, but the final goal is the same: to provide location transparency. Thus, the application sees only a global schema, but it does not in fact know whether a particular relation is locally or remotely available. The location transparency is a very important feature which brings a lot of advantages for the application developers, and it is well supported by the products.

[2]Due to general terms and conditions of the software licenses, we prefer not to use the real names of the products. Instead, we use nicknames which have no special meaning. Any similarity would be mere coincidence.

4.2 Transaction Management

When considering multiple, heterogeneous DBs and DML operations that may cross each of them, the transaction processing feature of the middleware receives special relevance. A complex query performed against heterogeneous data sources is usually handled as a distributed transaction. It divides it into small transactions and distributes the processing to the corresponding transaction systems of the affected DBMSs. The Two-Phase Commit (2PC) protocol [Gr78] is probably the most commonly used by the middleware systems in order to realize an atomic commit of a distributed transaction. We present the transaction management of the three products separately.

4.2.1 MIRACLE

An ORACLE Server copes with distributed transactions by means of a little modified 2PC protocol. Besides the usual *coordinator* and *participant roles* there is also the role of a *commit point (CP) node.* For the purposes of this paper, it does not matter how the CP node is defined. The processing of a distributed transaction is then performed as follows.[3] In the first phase of 2PC, the coordinator sends a *prepare-to-commit* message to each of the participants, but to the CP node. Thus, the CP node does not need to get *prepared.* The coordinator then waits for the *ready-to-commit* answers from the participants. If any participant does not respond in a predetermined timeout, then the coordinator sends *abort* messages to all participants, including the CP node. On the other side, if all participants have gotten prepared, the coordinator sends first of all a *commit* message to the CP node and waits for an answer. The normal or abnormal termination of the transaction on the CP node dictates now the fate of the whole distributed transaction. If it has committed its transaction, the coordinator sends *commit* messages to all participants and the distributed transaction is thus committed. Otherwise, the coordinator sends *abort* messages to the participants and the distributed transaction must be rolled back.

A distributed transaction which involves MIRACLE is handled by the ORACLE Server in a very similar way [Hu96]: the ORACLE Server always plays the role of the coordinator and MIRACLE is always considered to be the CP node. The reason for that is that MIRACLE does not support the *prepare-to-commit* operation. Hence, in the CP role MIRACLE must not get prepared, instead it must just commit/abort its

[3]This is a very simplified description of 2PC. The usual operations issued on behalf of e. g. the recovery manager, like write aheading the log records, forcing/no-forcing modified tuples into stable storage, discarding locks, etc. [Re97], are irrelevant in our context.

transaction like any usual transaction, without even knowing that it is in fact participating in a distributed transaction. Notwithstanding, due to the fact that there may be always one and only one CP role in the processing of a distributed transaction, only one MIRACLE may participate in a distributed transaction. Consequently, a distributed transaction in a MIRACLE environment may involve only one foreign (non-ORACLE) DB. This is of course a serious limitation, since it severely restricts the use of MIRACLE to solve the heterogeneity problem in a DB federation.

4.2.2 DJ

DJ uses the 2PC protocol to process distributed transactions too. First of all, DJ differentiates between two classes of applications [IBM97]:

- Class-1: applications which establish a connection with only one DB; and
- Class-2: applications which establish connections with several DBs.

In addition, on establishing a connection a parameter called SYNCPOINT can be set according to the following characteristics of distributed transaction processing:

- NONE: there is no coordinator in the 2PC. A commit message is sent to each participating DB. In the case of failures, the application must cope with the reestablishment of the DB integrity by itself.
- ONEPHASE: distributed transactions with updates on multiple data sources are not supported. If an application establishes a connection with DJ by using this parameter value, then no 2PC is processed outside DJ.
- TWOPHASE: distributed transactions with updates affecting multiple data sources are supported, i.e., the 2PC protocol is fully supported.

There are two typical configurations when employing DJ. First, the client establishes a Class-2-connection with SYNCPOINT TWOPHASE and defines which DBMS should be the coordinator. Thus, a foreign DBMS can play the role of the coordinator and DJ, in this case, assumes as a subcoordinator the coordination of its integrated sources of data. In the case of either a Class-1- or Class-2-connection with SYNCPOINT ONEPHASE, DJ is the coordinator and therefore responsible for the transaction processing.

In order that DJ can process 2PC with the integrated DBs, these must of course also support the 2PC protocol. Since this is not the usual case in all DBMSs, DJ employs another parameter in order to find out whether a data source supports the 2PC protocol (2PC-data-source) or not (1PC-data-source). On the basis of this information, DJ follows the subsequent rules:

- Read operations are always allowed with 1PC- and 2PC-data-sources.
- Updates on 1PC- and 2PC-data-sources cannot be mixed in a single transaction. In a SYNCPOINT ONEPHASE connection, a 1PC-data-source is only modified, if no other 1PC- or 2PC-data-source is modified either. In a SYNCPOINT TWOPHASE connection, no update on 1PC-data-sources is allowed.
- A distributed transaction cannot modify two 1PC-data-sources.
- The DJ's DB is always considered a 2PC-data-source.

On following these rules, DJ allows for an update on heterogeneous DBs only when the corresponding data sources support the 2PC protocol. Otherwise, such an update is simply rejected by DJ. Particularly, we consider the actions taken by DJ during the processing of a distributed transaction very reasonable, since the 2PC protocol may be executed in its entirety as long as the involved data sources understand the 2PC as well. Thus, with respect to transaction processing DJ is certainly more mature than the other products.

4.2.3 EDI/S

EDI/S provides no transaction manager by its own and therefore no support for 2PC. It offers the user the possibility to either automatically issue a commit after each SQL operation or the user decides when a commit should be processed [IBI97]. In both cases, EDI/S exploits the transaction managers of the integrated data sources in that it simply passes on the commit statements to the data sources. In case of an update operation affecting multiple sources of data, no kind of 2PC is available to control the transaction processing as a whole. EDI/S sends the update operations to the corresponding DBMSs and waits for the respective return parameters. If the update is successfully performed by all of them, the EDI/S server signals the success of the operation to the application. Otherwise, if at least one DBMS fails to perform the update, the EDI/S server simply passes on the received error code to the application. Hence, it is the responsibility of the own application to cope with the rollback of the update operations already successfully executed on the other involved DBs.

In order to exemplify the consequences of not supporting the 2PC protocol in a heterogeneous environment, the reader has just to imagine the well-known *debit-credit* transaction [GLPT76] being executed against two relations of two heterogeneous DBs. If EDI/S coordinates the processing of this distributed transaction and any of the participant transaction fails, no one may be completely sure about the final state of the DBs. Furthermore, the consequences may be really undesired: suppose the credit being done without the corresponding debit. At last, reestablishing a consistent state for the DBs after a transaction failure or a system crash may even be impossible.

4.3 DML and DDL Operations

All products support the execution of DML operations against heterogeneous data sources without problems. The applications can perform SELECT, INSERT, UPDATE, and DELETE operations in all DBs comprised by the federation. In turn, DDL operations are generally not allowed by DJ and MIRACLE. On trying to perform a DDL operation under MIRACLE, the application receives an error code back and the operation is rejected. Under DJ such operations cannot even be formulated, since by means of the nickname concept only integrated tables can be referenced. There is no construct available to formulate DDL operations, as for example, to create a table in a foreign DB. On the other hand, EDI/S does support the execution of DDL statements. Its activities are essentially controlled by the access rights the application has to the foreign DB, but they are not constrained by the middleware.

In the field of non-relational DBs, the operations supported by the middleware are not so diversified. The SELECT operation is supported by all products against non-relational DBs. However, the DML operations INSERT, UPDATE, and DELETE are supported only by EDI/S on IMS DBs. Therefore, with respect to the support of DDL and DML operations, EDI/S is certainly more powerful than DJ and MIRACLE.

4.4 SQL-Dialects and Pass-Through

It is a task of the middleware to cope with differences in the SQL implementations. The products we have analyzed solve this problem in basically two ways. Firstly, they provide a single SQL (either the standard or a particular dialect) API, and hence they mask the differences in the various vendor dialects of the language. Thus, the translation of SQL statements and return codes is assumed by the middleware. Secondly, they offer the so-called *pass-through* function. By means of this function, SQL statements can be sent to a particular DBMS directly, without that the middleware tries to interpret or translate such statements.

The three products differ in the functionality of the pass-through function. All of them support DDL operations. This is so because DDL operations are executed by the corresponding DBMSs without returning parameters which must be interpreted according to any specific format. However, it is worth mentioning here that the execution of DDL operations via pass-through is always followed by an implicit transaction commit on the corresponding DB by all three products. This means that an eventual rollback at the middleware level does not imply a rollback of DDL operations at the DB level. Therefore, by using the pass-through function to execute DDL operations, the distributed transaction processing features (2PC protocol) before mentioned (refer to Sect. 4.2) no longer hold.

In turn, the DML operation SELECT returns a result that must be formatted, and unfortunately MIRACLE cannot interpret such return values; it does not understand the statement and therefore does not know what has been read from the DB. On the other side, the other DML operations, INSERT, UPDATE, and DELETE, can be passed through by MIRACLE without problems. DJ and EDI/S are more flexible in this point; they support any kind of DML operation. This is one of the aspects that makes clear the differences between a middleware solution (as DJ and EDI/S) and a gateway software (as MIRACLE). COMMIT and ROLLBACK statements are supported only by EDI/S, which particularly is the most flexible of the products with respect to the pass-through function. Both statements are always either intercepted and interpreted by DJ and MIRACLE or, as previously mentioned, they are implicitly issued when processing DDL operations.

4.5 Referential Integrity

All products cope with this aspect in the very same way: they simply delegate to the single DBMSs the responsibility for checking and taking care of the referential integrity. They do not support the referential integrity between tables of a same DB, not even to mention between tables of heterogeneous DBs. However, if a DBMS rejects an operation due to a referential integrity violation, the corresponding error code is returned to the application program. Notwithstanding, if a particular DBMS does not support referential integrity, it is the application developer's responsibility to take care of it when issuing DML operations.

4.6 Security

Basically, it is possible to define four levels of security: the application must have login rights to the middleware (via usercode/password); it is possible to define access rights to the global schema of the middleware; the application must have login rights to the foreign DBMS (via usercode/password); and it must have access rights to the corresponding tables and tuples in the foreign DB. These four levels of security are well supported by all three products with irrelevant differences from each other.

4.7 Scalability

We have analyzed here how easy or difficult it is to append a new source of data to the global schema.

At this point, the difference between middleware and gateway comes again into the surface. As seen, a gateway provides a point-to-point connection to one type of DB. Thus, a heterogeneous environment employing MIRACLE demands a new gateway tailored to the new type of the DB which must be integrated into the global schema. In turn, on integrating a new DB, whose type pertains to the federation yet, a new instance of the available gateway must be configured.

On the other side, a middleware usually comprehends connection possibilities to several types of DBs. With DJ the user receives a complete software package which includes access to all supported DB types. However, the software components are turned on or off according to the license contract. Hence, in order to scale up the DB federation under DJ to comprise a new DB type, the corresponding module must be bought and turned on, and additionally, the configuration steps must be followed anew. Notice, however, that no new software component must be integrated. Similarly, EDI/S also comes as a complete software package, and single components are turned on or off according to license contracts.

4.8 Supported Data Sources and Platforms

The data sources and platforms supported by the three products are listed in Table 1. As can be seen, EDI/S supports the broadest range of data sources.

4.9 Query Optimization

The three products differ greatly from each other with respect to query optimization techniques. Query optimization plays a fundamental role when large quantities of data must be transported through a network that connects a DB federation. Particularly when executing joins against tables of heterogeneous DBs, a badly chosen execution plan can lead to a large, unnecessary overhead. We start explaining DJ's techniques.

4.9.1 DJ

The algebraic optimization in DJ is called *query rewriting processing*. The queries are transformed in a logical equivalent variation, in order to reduce I/O operations, sorts, communication overhead, etc. In the field of non-algebraic optimizations, DJ works with information about the data and their storage. On processing a query the optimizer decides which parts of the query can be processed by the data source and which should be done by DJ itself (*pushdown analysis*). Hence, I/O operations are saved and the communication overhead is reduced. The decision about whether the operation should be executed locally or by the remote source of data is taken based on a complex cost model. This cost model takes into consideration not only the costs of evaluating the operation but also costs for the communication between DJ and the remote data source for transferring the data or messages. The following factors are used by DJ on building its cost model: the relative CPU speed of DJ's and data source's machines, the relative I/O speed of both machines as above, the lowest communication rate between both machines, and the alphanumeric sort facility of the data source.

Additionally, DJ maintains a knowledge base containing information about how the query optimizers of foreign data sources work. Hence, DJ does not generate an execution plan for a query which would be also generated by the foreign DBMS; it does not generate those execution plans which would not be understood by the foreign DBMS; and so forth. Specifically for the synonyms (nicknames) created for the tables or views, DJ considers the following aspects: number of tuples in the referenced relation, number of pages occupied by each table, and lowest and highest value for the tables' columns. In addition, since the existence of indices influences the processing of queries, DJ also stores some information about them: the available indices in the foreign DBs and the access methods, number of levels of an index, number of leaves in an index, cardinality of an index key, and cluster behavior of clustered indices.

All these data and information are caught by DJ from the foreign DB. However, since some DBMSs do not put available this information, the own user can fill in the global catalog of DJ with such statistics. Furthermore, DJ can by itself generate the needed information by querying the sources of data accordingly.

4.9.2 MIRACLE

MIRACLE exploits some techniques used by the ORACLE Server for the query optimization (a *rule-based approach* and an *statistical approach*). However, MIRACLE does not maintain statistical data about the foreign DBMSs, as is the case in DJ. Thus, for a query affecting only foreign relations no kind of optimization is tried, and therefore MIRACLE shows unfavorable performance results in these cases (refer to Sect. 5.3).

4.9.3 EDI/S

EDI/S offers algebraic and non-algebraic optimization techniques, but only for joins. The algebraic optimization is called *WHERE predicate cloning*, and it consists of generating more conditions in the WHERE clauses whenever possible in order to specialize the search predicate. With respect to non-algebraic optimization, EDI/S maintains a relative cost model for the execution of joins. It is based on the type of the DBs and structures of the tables and columns. By

Table 1: Supported data sources and platforms.

Product	*Platforms*	*Data sources*
EDI/S	Windows (NT, 95, 3.x), OS/2, UNIX, MVS, CICS, VM, OpenVMS, Tandem, AS/400	ADABAS, Allbase, AAL-IN-1, CA-Datacom, C-ISAM, DB2, DBMS, DSM, Enscribe, FOCUS, CA-IDMS (/SQL), IMS, INFOAccess, Informix, Ingres, ISAM, Model 204, MSM, NOMAD, NonStop SQL, ORACLE, Progress, QSAM, Rdb, Red Brick, RMS, ShareBase, SQL/400, MS SQL Server, Stor-House, Supra, Sybase, System 2000, Teradata, TOTAL, ULTRIX/SQL, UNIFY, uniVerse(PICK)
DJ	*Server:* Windows NT, AIX *Client:* Windows (NT, 95, 3.x), DOS, OS/2, Macintosh, UNIX	DB2, EDA/SQL[a], IMS, Informix, MS SQL Server, ORACLE, Sybase, VSAM
MIRACLE	Windows NT, UNIX, MVS, CICS, IMS/TM, AS/400	ADABAS, DB2, EDA/SQL[a], FOCUS, IMS, IM"-AGE/SQL, Informan, Informix, Ingres, MS SQL Server, ORACLE, Rdb, RDMS, SAP, SESAM, Sybase, Teradata, VSAM

[a]By means of EDA/SQL, it is possible to access other data sources.

means of this cost model, EDI/S tries to determine the cheapest execution sequence for a join based on the costs to process each involved table, on the attributes' types, on the compatibility of columns, etc.

5 The Performance Evaluation

We have paid special attention to the performance of the products. In order to exhaustively evaluate the products' behavior in the face of different situations, we have implemented the AS^{3}AP Benchmark [TOB93]. AS^{3}AP is the ANSI SQL Standard Scaleable and Portable Benchmark for comparing relational DBMSs. Nevertheless, we have adapted the AS^{3}AP benchmark to fit into our purposes. Since we would like to test the performance of the gateways and not specifically of the DBMSs themselves, we have created heterogeneous DBs and kept this heterogeneity unchanged during the tests' execution. What we have varied then was the nature of the operations. In the following, we briefly introduce the benchmark, the environment where we performed the tests, and the results.

5.1 The AS^{3}AP Benchmark

We have chosen the AS^{3}AP benchmark for our performance tests due to its completeness in comparing relational systems with vastly different architectures and capabilities over a variety of workloads. The ANSI SQL Standard [Me90] DDL and DML are used in the specification of the DB tables and attributes and queries. It comprises two modules: the single-user tests and the multi-user tests [TOB93].

5.1.1 The Database Structure

The AS^{3}AP DB is composed of five relations. The *tiny relation* is a one column, one tuple relation used to measure overhead. All other four relations have the same average tuple width and the same number of tuples, and their size is scaleable. In the *uniques relation* all attributes have unique values. In the *hundred relation* most of the attributes have exactly 100 unique values and are correlated, so that selectivities of 100 and projections producing exactly 100 multi-attribute tuples are provided. In the *tenpct relation* most of the attributes have 10 percent unique values and thus selectivities of 10 percent are provided. Finally, the *updates relation* is customized for update operations.

5.1.2 The Single-User Tests

The single user tests are logically divided into operational issues and user queries.

Operational Issues

Most of the operational issues is of little interest when thinking about the performance of DB gateways. They include loading the DB from disk or tape, a backup procedure, the time to build indices, and a table scan. This latter is the only one of interest for us:

Table scan. This operation tests sequential I/O performance; it searches a tuple that in fact does not exist via an attribute which has no index built on it.

Test Queries

The test queries include output tests, selections, joins, projections, aggregates, and updates.

Output mode queries. The three different output modes that a query may use are tested, namely, the results may be retrieved to the screen, stored in a file, or in a DB relation. These queries are listed in Table 2 [TOB93]. Comparing the measurements of these queries with the table scan above provides a precise estimate for the time required to format, display, and store the result of a query.

Table 2: Output mode queries.

Query name	*Description*
o_mode_tiny	selection on tiny relation
o_mode_1K	selection of 10 tuples
o_mode_10K	selection of 100 tuples
o_mode_100K	selection of 1000 tuples

Selections. The two most important factors influencing the performance of query processing are controlled by the

AS^{3}AP benchmark: the storage organization of the relation and the selectivity factor of the query. The selections are listed in Table 3 [TOB93]. In particular, the *variable_select* query tests the ability of the query optimizer to correctly choose between a scan or the use of an index at run time.

Table 3: Selection queries.

Query name	*Description (index)*
sel_1_cl	1 tuple (clustered)
sel_1_ncl	1 tuple (sec. hashed)
sel_100_cl	100 tuples (clustered)
sel_100_ncl	100 tuples (B-tree sec.)
sel_10p_cl	10% tuples (clustered)
sel_10p_ncl	10% tuples (B-tree sec.)
variable_select	range predicated on variable

Joins. The joins test how efficiently the system makes use of available indices and how query complexity affects the relative performance of the DBMS. The join queries are presented in Table 4 [TOB93]. The *join_2* queries test the performance of ad-hoc queries. The use of the available indices in the join algorithm is tested by varying the join attributes and thus the types of indices. Finally, query complexity has been modeled by increasing the number of tables referenced.

Table 4: Join queries.

Query name	*# tables*	*Index available*	*Retrieved*
join_2	2	no index	1 tuple
join_2_cl	2	clustered	1 tuple
join_2_ncl	2	non-clust. hashed	1 tuple
join_3_cl	3	clustered	1 tuple
join_3_ncl	3	non-clust. hashed	1 tuple
join_4_cl	4	clustered	1 tuple
join_4_ncl	4	non-clust. hashed	1 tuple
join_1_1pct	2	clustered	1% tuples

Projections. The most expensive part of a projection is the elimination of duplicate tuples when projecting on non-key attributes. Duplicate elimination is usually done by sorting. The projection queries, listed in Table 5 [TOB93], test mainly the DBMS's sort utility. In particular, the first query tests how efficiently two different data types are handled in a complex operation such as sorting.

Table 5: Projection queries.

Query name	*Description*	*Selectivity*
proj_100	on signed int and on variable length char attribute	100 tuples
proj_10pct	on decimal attribute	10% tuples

Aggregates. Different kinds of aggregate tests are provided (Table 6 [TOB93]). The first computes the minimum of a key value. The second computes the minimum value of a key partitioned into 100 partitions. The *info_retrieval* query tests whether systems can use bit or pointer intersection algorithms for indices to avoid a relation scan to evaluate complex predicates. The other queries test report generation features, such as subtotal and total calculations.

Updates. Essentially, the update tests check both integrity and performance (Table 7 [TOB93]). To check integrity, one query attempts to append a tuple with a duplicate key value. A second attempts to violate referential integrity by updating a foreign key. Performance is tested by measuring the overhead involved in updating each type of index. Finally, single-tuple updates and bulk updates are also considered.

Table 6: Aggregate queries.

Query name	*Description*	*Result*
scal_agg	min(key)	1 val
func_agg	min(key) grouped by name	100 val
info_retrieval	sel with complex predicate, min(key)	1
simple_report	sel avg(x) where x in (sel 10%)	1
subtotal_report	10% sel on view, min(a), max(a), avg(a), cnt(b), grp by code, int	100
total_report	10% sel on view, min(a), max(a), avg(a), cnt(b)	1

5.1.3 The Multi-User Tests

The four different multi-user tests model different workload profiles [TOB93]. For these tests, a number of processes are forked concurrently, each running a single script. These processes then simulate concurrent users. In particular, the number of users is determined as a function of the DB size. The tests are explained in the following.

Information retrieval (IR) test. All users execute a single-row selection query on the same relation using a clustered index. The isolation level is 0 (browse access).

Mixed workload IR test. One user executes a cross section of ten update and retrieval queries (refer to Table 8 [TOB93]), and all others execute the previous IR test.

Table 7: Update tests.

Test name	*Description*	*Tuples upd*
append_duplicate	ins duplicate value	app 1, del 1
refer_integrity	ins invalid frgn key	ins 1, del 1
update_key	ins mid, last key val.	ins 2, mod 2
update_btree	1 upd of B-tree idx	ins 1, mod 1
update_alpha	1 upd of idx on alpha fld	ins 1, mod 1
bulk_update	bulk ins in mid key range	ins 1000
bulk_modify	bulk upd in mid key range	mod 1000
bulk_delete	bulk del in mid key range	del 1000

OLTP test. All users update a single row on the same relation with level 3 isolation (repeatable reads). The operation consists of randomly selecting a single row via a clustered index and updating a non-indexed attribute.

Mixed workload OLTP test. One user executes the cross-section queries (Table 8 [TOB93]) and the others execute the previous OLTP test.

The throughput, as a function of the number of concurrent DB users, can be measured by the OLTP and IR tests. The mixed workload IR test measures the degradation in response time for a cross section of queries caused by system load. The system ability to dynamically provide different isolation levels is also tested.

Table 8: Cross-section queries.

Query name	*Description*	*Isolation*
o_mode_tiny	overhead query	3
o_mode_100K	retrieve 1000 tuples (100 Kb)	2
select_1_ncl	1 tuple sel on 3 isol. levels	0, 1, 2
simple_report	sel-join-aggregate	2
sel_100_seq	sel 100 clust. into temp	3
sel_100_rand	sel 100 rand into temp	3
mod_100_seq	upd 100 seq tuples, abort	3
mod_100_rand	upd 100 rand	3
unmod_100_seq	undo prev seq upd	3
unmod_100_rand	undo prev rand upd	3

5.2 The Environment

When building the environment for our tests (Fig. 2), we have tried to reflect the same environment that we have in MENTAS. As shortly mentioned before, the DBs to be integrated in MENTAS are from IBM DB2 and from ORACLE. Thus, we have created two identical DBs (40 MB each) from these vendors – ORACLE Version 7.3.2 for RS/6000 and DB2 Common Server Version 2.1 for RS/6000. The products' Clients were put on a same machine, the client. The benchmark operations were sent from this client to the corresponding Servers of the products, which in turn were located in a same machine either, the server. The client is an HP 9000 Series 700 Model B160L with 192 MB of main memory and running HP-UX 10.20. The server is an IBM RS/6000 Workstation Model 3BT with 128 MB of main memory and running AIX 4.1.4.0. The versions of the three products are: DJ Version 2.1 for AIX, EDI/S Version 3.2 for UNIX, and MIRACLE Version 4.0 for IBM DRDA for RS/6000.[4]

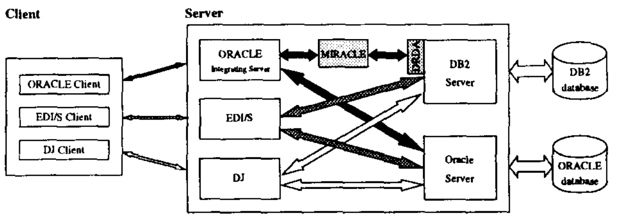

Fig. 2: A simplified overview about the environment.

The statements affecting a single table were performed twice, once against each kind of DB. We have measured both access times and show in the forthcoming sections the average time between both measurements. By all join statements, we have distributed the tables over both DBs. The joins have the following structure:

[4]It is important to notice here that there is a version of MIRACLE tailored for DB2 which does not need to use DRDA for the communication with the DB2 Server. However, this version is only available for DB2 under MVS. Since our DB2 DBs in MENTAS are all for RS/6000, we were not interested in testing this version of MIRACLE. Due to the fact that this version accesses the DB2 Server directly, we believe it would have shown better performance results than this one via DRDA.

```
SELECT tab1.attr1, ..., tab1.attrn, ...,
       tabm.attr1, ..., tabm.attrn
  FROM tab1, ..., tabm
 WHERE tab1.attrx = tab2.attrx AND tab1.attrx = tab3.attrx
   AND ...
   AND tab1.attrx = tabm.attrx AND tab1.attrx = value;
```

Table tab_1 was placed in one DB whereas the tables tab_2 to tab_m were placed in the other DB. We have run all join statements twice switching the location of the tables in the second run.

5.3 The Performance Results

In the total we have generated 85 graphs showing the performance of the three products compared to each other in the most different ways. Of course we cannot show all of them here due to space limitations. We have then chosen 8 of the most significant graphs, and we present them here. In these graphs, we have maintained the results of DJ in the X-axis constant and shown in the Y-axis the percentage, i.e. the *relative* variation, that MIRACLE and EDI/S are better or worse than DJ. In the following, when explaining the graphs, this representation will become clear.

5.3.1 The Single-User Tests

Operational Issues

Table scan. The run sequence of the benchmark establishes that the table scan must be performed intercalated with the select operations. Thus, we have not generated a separate graph for the table scan. Instead, we have put the results together with the selects shown in Fig. 4. The results of the table scan are shown in the third blocks from left to right. The most interesting fact we have perceived with the table scan, and which have proved true in other similar operations along the benchmark, is that DJ does not perform well when operating on a non-indexed attribute. As seen, the table scan searches a tuple that in fact does not exist via an attribute which has no index built on it, and in this case MIRACLE and EDI/S have performed 43% better than DJ.

Test Queries

Output mode queries. Fig. 3 shows the results for the output mode queries. As seen, there are four operations (refer to Table 2) in which the results are retrieved to the screen, stored in a file, or in a relation. DJ is almost always the fastest of them to format, display, and store the results of a query. Notice that DJ was the slowest in the table scan test previously discussed. In these output queries, we can also notice that EDI/S can work well with large amounts of data, and it is sometimes better than DJ (*o_mode_100K(file)* and *o_mode_100K(screen)*). This means that the time EDI/S needs to fetch the tuples is compensated by faster routines to format and present the results. In turn, MIRACLE is generally better than EDI/S, but, on the contrary, it shows really bad performance results when manipulating large amounts of data. This can be observed specially when the results are written in a file and shown on the screen.

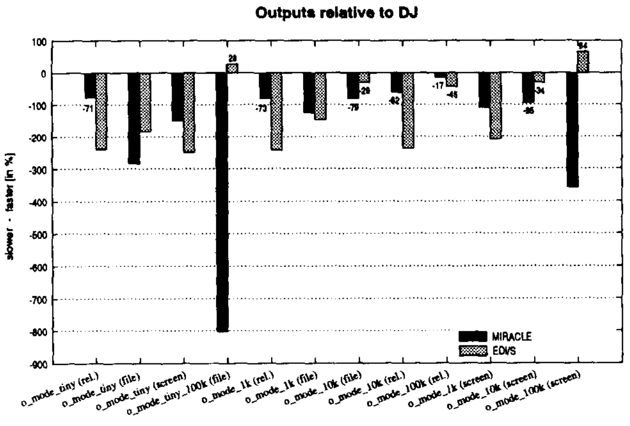

Fig. 3: Output mode queries.

Selections. Fig. 4 illustrates the performance results for the select operations (refer to Table 3). As before, DJ is almost always better than the others. As already mentioned, the table scan is performed better by MIRACLE and EDI/S than by DJ. Interesting here is that this is the only operation on a non-indexed attribute. Thus, DJ can exploit the existence of indices very well, but it performs worse than the others in the absence of indices. Furthermore, we believe that the reasons for DJ to work so well in the presence of indices are due to the statistics about the indices of all DBs that it maintains internally (refer to Sect. 4.9). Such detailed information is neither available to EDI/S nor to MIRACLE. Another aspect that can be noticed is the behavior of EDI/S on executing the *variable_select* operation. As explained before, this operation tests the ability of the optimizer to correctly choose between a scan or the use of an index at run time. There are two variations of this operation. In the *variable_select(low)* variation, an index should be used, and in the second, *variable_select(high)*, the table should be scanned. Since both operations are executed in order, we believe that EDI/S had executed a scan on the table during the first operation (it has shown worse times) instead of using an index. In turn, in the second operation the table was memory resident, and hence the scan was performed very fast.

Joins. DJ beats the others in the processing of join operations (refer to Table 4). Fig. 5 shows the performance results. DJ is up to 400% better than EDI/S and up to 60,000% better than MIRACLE in processing joins. Certainly, these results are due to the very refined query optimizer of DJ. An interesting aspect here again is the result of the query *join_2*. This is the only join which has been performed on an attribute which has no index built on it. In this case, EDI/S was even better than DJ. Particularly, the extremely bad performance of MIRACLE in this case leads us to the conclusion that the middleware solution to the heterogeneity problem is much more appropriate than the gateway approach. Furthermore, since the integration of heterogeneous schemas is mainly done by means of join

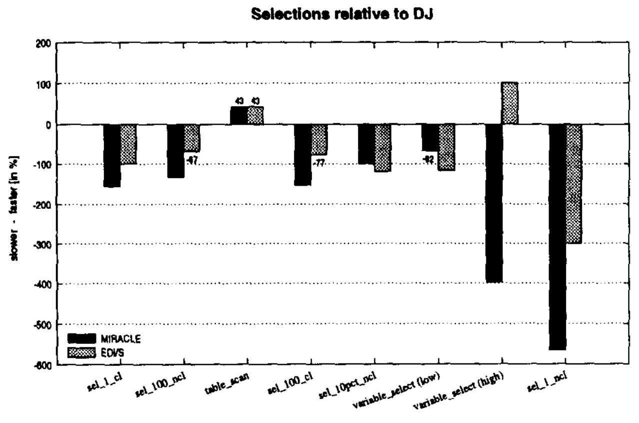

Fig. 4: Selections (including table scan).

operations, a well-designed query optimizer plays a kernel role in the solution to the heterogeneity problem because it greatly influences the performance.

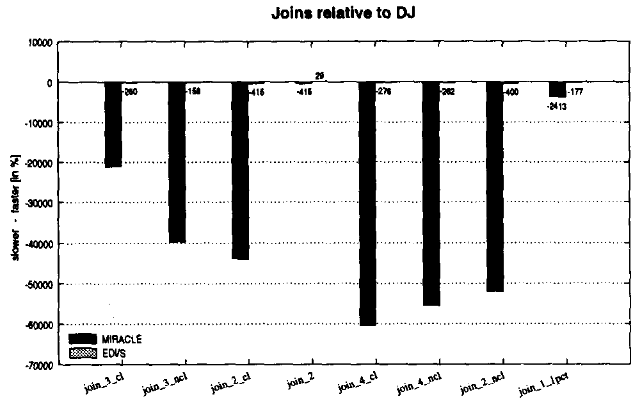

Fig. 5: Joins.

Projections. In the projection queries (refer to Table 5), EDI/S is faster than DJ and MIRACLE (Fig. 6). In the query *proj_100*, where 100 tuples (0,1% of the total) are read, EDI/S is about 200% better than the others. With respect to DJ, the problem is again the absence of an index for this query. In turn, MIRACLE is particularly slow to project (and thus to sort) on decimal values (query *proj_10pct*). In this case, DJ performs almost equal to EDI/S.

Aggregates. The results we have obtained when running the aggregate queries (refer to Table 6) were very interesting and sometimes surprisingly. In the aggregate query *info_retrieval*, DJ, as previously, has maintained its superiority when performing select operations (Fig. 7) in comparison to EDI/S and MIRACLE. The *subtotal_report* query has shown us that DJ can work very well with views (up to 700% better than the others). This is because this view is built through a join on two tables, and DJ has yet proved to be the best of them when joining relations. On the other hand, the *scal_agg* and *func_agg* queries are very simple ones employed to calculate the minimal value of an attribute which is the primary key of the relation. In fact, as explained in Sect. 4.9, DJ already maintains the minimal value of all primary keys in its own internal statistics for

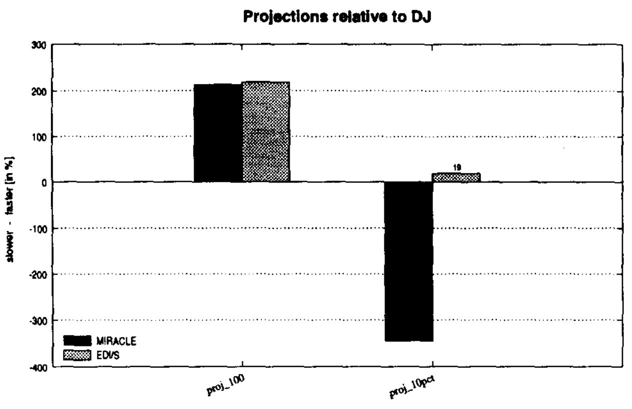

Fig. 6: Projections.

query optimization. Surprisingly, EDI/S and MIRACLE have performed up to 400% better than DJ to calculate these aggregate functions. We simply do not have a plausible explanation for that.

Furthermore, all three products have had problems with the *total_report* query (refer to [TOB93] for details on it). We have tried to overcome this problem by changing and newly running the query several times in order to locate the error. In fact, it is not an error of the gateways themselves but of the DB2 DB. The problem stays in the evaluation of the aggregate function $avg(a)$ by DB2, where a is a signed integer in the range $\pm 5 * 10^8$; it always returned an arithmetic overflow (SQLSTATE = 22003). Without this average calculation, the query can be executed successfully. In turn, DJ could not perform the *simple_report* query whenever it should be executed against the ORACLE DB. The ORACLE Server always returned a "*non-critical system error*" due to a column number out of range (SQLSTATE = 58004). Interesting is that EDI/S and MIRACLE have both executed the same query successfully. These are the reasons why we do not have performance results for the *total_report* and *simple_report* aggregate queries.

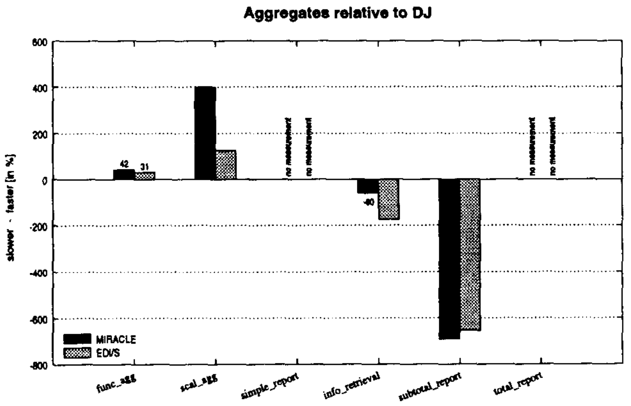

Fig. 7: Aggregates.

Updates. MIRACLE performs generally better than DJ and EDI/S in the integrity checks (refer to Table 7). Fig. 8 shows that EDI/S and MIRACLE are up to 40% faster than DJ when checking the integrity during update and delete operations. However, on trying to insert a double value for a primary key, both are much slower than DJ (up to 150%). The reason for this stays probably in the statistics that DJ maintains and exploits about all primary keys.

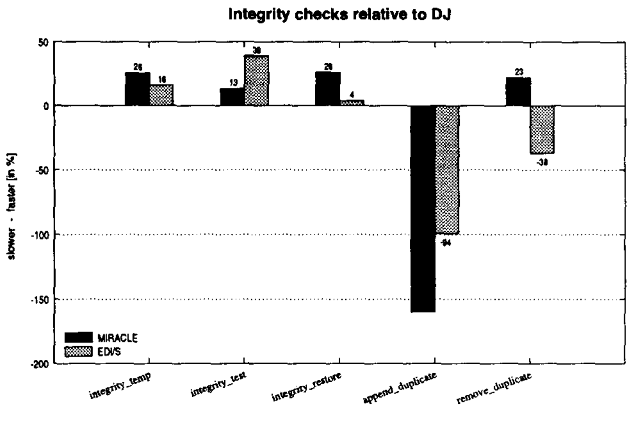

Fig. 8: Integrity checks.

As presented in Table 7, the update operations are further divided into updates affecting one tuple, *single updates*, and those affecting a number (1 000) of tuples, *bulk updates*. Fig. 9 presents the performance results for the single updates. As with the integrity checks, EDI/S and MIRACLE are faster to perform delete and update operations. In this case, both perform very similar. On the other hand, as before DJ shows better performance results for the insert operations than EDI/S and MIRACLE. Interesting here is that all insert operations are realized on indexed attributes. Similarly to the single updates, the bulk updates confirm the before mentioned observations (Fig. 10). EDI/S and MIRACLE are better when coping with update and delete operations, whereas DJ is the best one when inserting tuples in a relation.

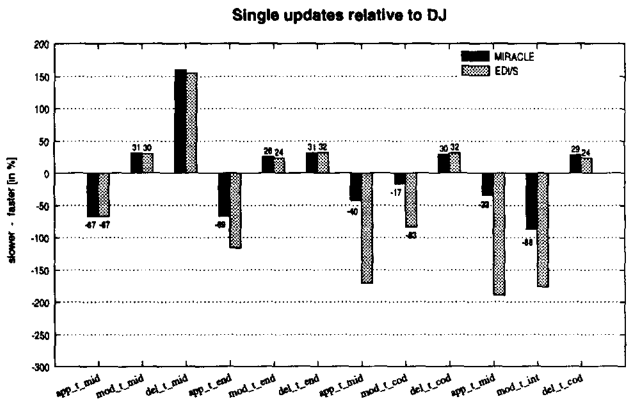

Fig. 9: Single updates.

5.3.2 The Multi-User Tests

We have performed the OLTP and IR tests (refer to Sect. 5.1.3) with 10 concurrent users and the mixed workload OLTP and IR tests with 11 users, 10 executing the corresponding original test and 1 executing the cross-section queries (refer to Table 8). The system has been firstly warmed up with the execution of the

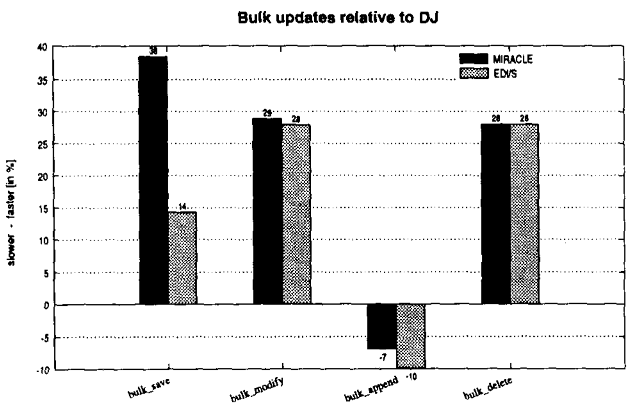

Fig. 10: Bulk updates.

queries for 15 minutes, and then we have started measuring the throughput during 5 minutes. Since these tests are performed on a single table, we have run them twice, once against a table of an ORACLE DB and once against a table of a DB2 DB. The throughput as a function of the number of concurrent DB users is listed in Table 9.

Table 9: Throughput in the multi-user tests.

	ORA	*DB2*	*ORA*	*DB2*
	IR		*mixed IR*	
DJ	6,946	25,841	3,098	24,142
EDI/S	2,881	2,679	2,529	2,365
MIRACLE	16,044	9,380	10,279	7,524
	OLTP		*mixed OLTP*	
DJ	7,396	13,093	5,483	12,055
EDI/S	3,125	2,989	2,886	2,672
MIRACLE	7,976	9,077	4,569	7,636

DJ has maintained its superiority by the processing of select operations, and has achieved a very high throughput on DB2. The throughput of MIRACLE on DB2 is considerably higher than that of DJ on an ORACLE DB. EDI/S has performed very poorly when compared to the other two. On the other side, the OLTP tests have revealed some interesting facts. EDI/S has still performed poorly and shown us that it can cope a little bit better with update operations than with selects. Against an ORACLE DB, DJ has even increased its performance in the processing of the updates, but its performance was drastically reduced when the updates were executed on a DB2 DB. Thus, it can perform selects much better than updates. Further on, the throughput achieved by MIRACLE when executing the updates against a DB2 DB was higher than against an ORACLE DB.

6 Conclusions

We have critically evaluated three DB middleware products and compared them with each other with respect to several aspects, inclusive performance. The most important differences between the products can be summarized as follows. Transaction management is well supported by DJ which fully implements the 2PC protocol to process distributed transactions. EDI/S supports a really broad range of DDL and DML operations and many platforms and data sources. DJ employs very refined query optimization techniques whose benefits can be clearly perceived in its performance. The AS^{3}AP benchmark has allowed us to evaluate the performance of the products over a variety of workloads and through the most different perspectives. DJ has shown problems whenever operating on non-indexed attributes, but it is unbeatable when performing select and, specially, join operations. The processing of views is also very well supported by DJ.

The global schema in MENTAS is mapped to the original data sources mainly via views. These views either select certain attributes of the relations or join existing relations to capture and relate information. Therefore, the efficient processing of selects and joins plays a crucial role in MENTAS. Thus, we have committed for DJ in MENTAS.

Acknowledgments

We are greatly in debt with Alexander Jassner and Jochen Rütschlin for always providing the necessary technical support during all phases of our tests.

References

[Bo92] Bobrowski, S. *ORACLE7 Server - Concepts Manual.* ORACLE Co., 1992.

[Gr78] Gray, J.N. Notes on Database Operating Systems. In: R. Bayer, M. Graham, and G. Seegmueller (Eds.), *Operating Systems: An Advanced Course*, LNCS 60, Springer, Berlin, 1978. pp. 393–481.

[GLPT76] Gray, J.N., Lorie, R., Putzolu, F., and Traiger, I.L. Granularity of Locks and Degrees of Consistency in a Shared Data Base. In: *Proc. IFIP Working Conf. on Modeling in DBMS*, Freudenstadt, 1976. pp. 365–394.

[Hu96] Hughes, K. *ORACLE Transport Gateway - Installation and User's Guide for IBM DRDA for RS/6000* (Release 4.0). ORACLE Co., 1996.

[IBI97] Information Builders Inc. *EDA/SQL Manuals.* Information Builders Inc., 1997.

[IBM97] IBM Co. *DB2 DataJoiner: Administration Guide and Application Programming* (Version 2 release 1). IBM Co., San Jose, 1997.

[Me90] Melton, J. (Ed.) *Database Language SQL 2.* ANSI, Washington, D.C., 1990.

[Re97] Rezende, F.F. *Transaction Services for Knowledge Base Management Systems - Modeling Aspects, Architectural Issues, and Realization Techniques.* infix Verlag, DISDBIS 35, 1997.

[TOB93] Turbyfill, C., Orji, C., and Bitton, D. AS^{3}AP: An ANSI SQL Standard Scaleable and Portable Benchmark for Relational Database Systems. In: J. Gray (Ed.), *The Benchmark Handbook for Database and Transaction Processing Systems*, Morgan Kaufmann, 1993.