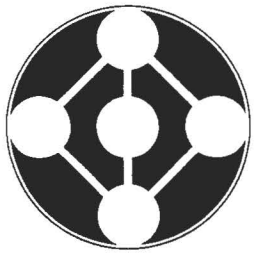

Alternative Technologies

Enterprise Database System Requirements

David McGoveran
Alternative Technologies
13130 Highway 9, Suite 123
Boulder Creek, CA 95006
Telephone: 408/425-1859
FAX: 408/338-3113

I. Introduction

Enterprise-wide information systems (IS) planning is increasing in importance with the development of a world market. Even small companies must compete in the world market for resources and market share. This rapidly developing situation has placed new and potentially costly business requirements on IS. At the same time, information technology is developing at an unprecedented rate. RDBMS requirements of today's enterprise information systems are discussed in the broader context of enterprise IS goals. The key issues being addressed by RDBMS vendors are examined, including reliability, scalability, cost effectiveness, and openness.

II. RDBMS Reliability

A. Understanding reliability

What is reliability in a RDBMS? First, it is closely related to availability. A RDBMS which has only continuously available features is certainly reliable. However, it is also possible for a RDBMS to be reliable if loss of availability can be planned and managed. Second, RDBMS reliability is related to quality. For an application to be reliable, the quality of data on which it relies must be controlled and guaranteed.

A *reliable* RDBMS is, first and foremost, predictable. It is not necessarily one that is fault tolerant, although high availability is certainly one aspect of reliability. End-users must be able to rely on the RDBMS to deliver timely data, without corruption. Administrators must be able to rely on the RDBMS with respect to capacity planning and other database administration tasks. Data integrity features, non-conflicting data definition, data manipulation, security and maintenance operations, transaction isolation, and high concurrency are key aspects of reliability.

B. Evaluating reliability

When evaluating a RDBMS for reliability, a number of issues must be addressed including continuous operation, robustness, integrity, and non-interference between operations. Each of these is examined in turn below.

By *continuous operation*, we mean that the system should reduce or even eliminate both planned and unplanned outages. Often, a RDBMS must be taken off-line in order to run some utility. Utilities for database loading, backup, recovery, integrity validation, index reorganization, etc., should all be executable on-line. If a failure occurs during such maintenance operations, the utility should not have to be restarted from the beginning of the operation. For example, it might perform periodic checkpoints or even full journaling so that utilities could continue processing from the point of failure. Other capabilities such as on-line and automatic archival of full log files, automatic restart (i.e., no operator intervention), and controllable system restart times are also important; these minimize operations which can require that the RDBMS be taken off-line.

By *robustness*, we mean that the RDBMS reduces the importance of any particular failure and recovers from it transparently. The RDBMS should provide both read and write access to data regardless of the circumstances, including hardware system or component failure. Systems which provide such high availability are said to be fault tolerant. Fault tolerant systems often rely on some various forms of redundancy, including redundant hardware, data, and processes to eliminate any single point-of-failure.

Hardware redundancy takes many forms, including completely redundant systems, redundant processors (either loosely coupled or tightly coupled), and redundant storage media (e.g., disk mirroring). When data is made redundant in a relational RDBMS it is said to be replicated. Hardware redundancy can minimize the time it takes to restart application software in the event of hardware failure. It is also possible for the software to make use of hardware redundancy, detect failures, and restart automatically. The user would not be able to detect that a failure had occurred if the ability to switch to back-up hardware were sufficiently sophisticated.

Another important aspect of providing high availability is distribution. By spreading out the location of software processing and data, the impact of any particular site or hardware failure can be reduced. Thus the amount of (hopefully transparent) software recovery which must take place following a failure can be reduced. Client/server architectures are a form of this kind of software distribution. They separate application-specific portions of an application from those portions that can be handled by sharable services and resources. Both the clients and the sharable services can be further distributed. In the case of a RDBMS, the distribution can entail

distribution of both RDBMS processes and data. When a RDBMS allows data to be distributed while providing the user with a stable view of the data, we say that the RDBMS supports location transparency (also known as location independence).

Regardless of where the data is located or how it is redundantly stored, the RDBMS provides a single logical location to the user, even when the physical location or the manner of redundancy is altered. For example, in addition to being replicated, a relational table might be split into multiple parts called fragments. The RDBMS might support either horizontal fragmentation meaning that row subsets of the table are physically distributed or vertical fragmentation meaning that column subsets of the table are physically distributed. It is also possible for multiple processors (perhaps widely separated in a physical sense) to share in query processing. This distribution of resources not only increases system availability, but may be intended to improve performance in certain circumstances.

Distribution of data introduces major technical problems for a RDBMS. For example, distributed transaction management in a system with multiple processors requires coordination among them for transaction commit, transaction rollback, and recovery. Replication requires an ability to synchronize redundant copies so that data integrity is maintained -- each user should perceive a single logical database rather than multiple, independent copies which contain different data.

Of course, it is sometimes possible to divide data among applications, users, or transactions so that independent databases can be maintained. A database with this property is said to be segmentable. It might still be desirable to administer a single logical database, however.

Even if a system supports continuous operation and overall high availability, throughput can suffer from *interference* between various logical operations. For example, data and security definitions are often incompatible with the ability to write to affected database objects and sometimes with the ability to read those objects as well. It is particularly important that an enterprise database system support mission critical applications by eliminating operation interface. In particular, the database should permit updates which do not interfere with reads and vice-versa. Utility operations on large tables such as backup, load, index creation, and index reorganization may require that the table be taken off-line. Similarly, physical database allocation and reorganization, space management, log archiving, and system restart should have a minimal impact on applications.

A major factor in controlling operation interference is a good concurrency control system, which is intended to manage concurrent transactions while insuring isolation between them. Typical concurrency control systems are pessimistic, which means that they use locks to provide isolation. Duration of lock waits, the number of locks acquired, lock granularity (row, page,

or table), and when locks are released all affect the degree to which concurrent transactions interfere. Optimistic concurrency control systems assume that interference will be exceptional and so do not hold locks. Typically, optimistic schemes maintain multiple copies of the data (multi-versioning or multi-generations) and then resolve conflicts at commit time by rejecting one or more of the conflicting transactions.

Regardless of the concurrency control mechanism used, it is important that the database system provide a clear definition of, and some control over, the degree of isolation available. Traditionally, five levels of isolation (such as dirty read, cursor stability, and repeatable read) were defined based on a pessimistic concurrency control scheme. However, when a database system uses a different scheme or permits schemes to be intermixed, the definition of these levels of isolation and the corresponding ways in which integrity can be lost may differ from the traditional one. It is important that users understand these differences when selecting a database system for enterprise use, since a loss of integrity can affect an entire organization.

An RDBMS always needs to support integrity, but for enterprise RDBMS *automatic integrity enforcement* takes on a renewed importance. It may not be unusual for a personal or departmental RDBMS to support only a few applications and a smaller database. With a relatively small database, relationships between tables are easy to identify. If the applications which use the database are fixed in number and function and if ad-hoc update is not needed, integrity can be managed to some extent in the application. Of course, this practice is deprecated, but it is all too common. With automatic integrity enforcement, all integrity rules are defined in and enforced by the RDBMS so that neither the mistakes of ad-hoc users nor bugs in application programs can corrupt the database.

Integrity issues have traditionally been classified as belonging to one of five types: entity, referential, domain, column, or user-defined. *Entity integrity* demands enforcement of primary key discipline and is usually accomplished by declaring a column or set of columns in a table to be the primary key. *Referential integrity* enforces certain relationships between tables in which one table must reference another. A standard set of actions has been defined (these are not exhaustive) which permit the user to declare both the relationship and how update, insert, and delete operations are to be handled. *Domain integrity* enforces rules regarding the values that are permitted in a domain and what operations are permitted on those values. *Column integrity* does the same thing for individual column, which are themselves drawn from some domain of permissible values. Column integrity is usually enforced by specifying declarative constraints (uniqueness, non-null, values ranges, etc.) when the table is created. Since domains are usually not supported by RDBMSs, there is no standard way in which domain integrity is handled. *User-defined integrity* relates to business rules. Most RDBMSs enforce user-defined integrity through a database trigger

mechanism.

V. RDBMS Scalability

A. Server "Size"

A *scalable* RDBMS is essential in any business with changing needs and workloads. It permits planning for growth and technology change. Scalability is often understood in terms of *linearity*. For example, if the workload doubles, performance should be reduced to no less than about fifty percent. Similarly, adding twice as much computing power (memory and CPU) should roughly double performance or perhaps the number of concurrent users.

An RDBMS can be scalable either horizontally or vertically. Horizontal scaling refers to the ability to add (or remove!) database servers. It requires support for distribution, portability, and cross-platform interoperation. Vertical scaling refers to the ability to increase (or decrease!) database server computing power, either within a given family of computer hardware and operating systems or, if necessary, using a completely different platform. RDBMS vendors have usually focused on upward scalability. Downward scalability is just as important, since it permits the company to downsize more easily and to eliminate unnecessary computing resources.

B. Server Interoperation

Support for horizontal scaling requires that servers interoperate transparently. As the workload needs of the enterprise increase, additional servers can then be added without disrupting applications. The distribution of requests to servers can be managed by either the application, a gateway or name service, or by servers. Request distribution by the application places a burden on developers and system managers. Request distribution by a gateway usually requires additional hardware which can become a bottleneck as well as a single point of failure. Both of these forms of distribution requires RDBMS support for remote server access.

In server-to-server request distribution, each server in a group is known to all the others and can forward requests as necessary. If a database server supports distributed query processing and shared access to data, automatic workload distribution becomes possible. Such interoperating servers are said to be cooperative servers.

In an enterprise-wide distributed computing environment, it is likely that cooperative servers will be arranged in groups. These groups of servers can then behave as though they were a single server, providing the benefits of horizontal scaling. In addition, groups of servers may be interconnected and form distributed server groups.

C. Transparent Distribution

In principle, distributed server groups and cooperative servers can lead to enterprise wide data sharing. At the same time, fault tolerance, availability, and reliability can be improved. Users can distribute or consolidate data as needed.

For distributed server groups to interoperate transparently, the database must be fully distributed. This means that distributed transactions, a distributed data dictionary and global naming scheme, and platform (network, hardware, and operating system) transparency must be supported by the RDBMS, in addition to distributed query processing (distributed join optimization), site autonomy, no single point of failure, and fragmentation and replication independence.

In order for this to work well, the RDBMS must be able to take advantage of both loosely coupled and tightly coupled multi-processor systems. A key issue is cache coherence or consistency. The method by which cache coherence is maintained has an affect on which levels of transaction isolation can be supported, on the efficiency of scaling, on the robustness of the system (a single point of failure can be introduced inadvertently), and on recovery mechanisms.

When symmetric multiprocessing (SMP) is used, processors can share access to a common cache. This means that cache coherence can be maintained by some mechanism of mutual exclusion, preventing more than one processor to have access to the same portion of cache at any given time. A good cache management scheme must provide mutual exclusion without the mechanism becoming a bottleneck.

In a loosely coupled processor scheme, cache coherency is more difficult to maintain efficiently because memory is not shared. For this reason, mutual exclusion must be made known to all participants. If multiple copies of cache are used, there can be only one valid copy of any portion of cache at any time. The acceptable methods of maintaining cache coherence depend on how loosely coupled processors are interconnected and on what mechanisms exist for communicating global information. Typically, loosely coupled processor systems provide some form of distributed lock management. If a network is involved, locking overhead can be extremely high.

A number of other facilities can provide significant benefit in a cooperative server environment. Support for triggers and alerters can provide a means for user-defined server-to-server communications. Transparent two-phase commit and declarative integrity reduce the amount of effort involved in re-distributing data among servers. Without these, application code must be modified to take into account changes in which servers are accessed and for what data. With declarative integrity, the distributed RDBMS system catalog can be accessed to determine how integrity is to be enforced since this information need not be

dependent on data or table location. Similarly, TP monitor support can provide integration of cooperative servers with foreign data sources and existing applications.

VI. RDBMS Cost Effectiveness

Within the RDBMS industry, RDBMS cost of ownership is often obtainable from TPC benchmarks. These benchmarks report $/TPS, which include hardware and software ownership and support costs over a five year period. A lower cost RDBMS enables IS managers to be less concerned about performing a "cost/benefit" or "break even" (i.e., determining the point at which costs outweigh benefits) analysis.

A *cost effective* RDBMS can have an important beneficial impact on all the costs of data processing. The many benefits of relational systems are often lost because the overall cost of ownership of the RDBMS is too high. Ease of use, ease of maintenance, and ease of application development are all important in this regard since they reduce costs due to training, day-to-day operation, and maintenance. In addition, both improved performance and the elimination of data corruption errors should be goals in moving to relational applications. All too often, mis-design of relational applications leads to poorer performance than was found in non-relational applications. The RDBMS should encourage proper design of relational and distributed (e.g. client/server) applications by providing the appropriate facilities.

A. Application Design, Development, and Maintenance Costs

Application design, development, and maintenance costs have a strong impact on RDBMS cost effectiveness. The relational model was designed to reduce these costs: as a result, it is important that the RDBMS adhere closely to the relational model. Support for logical data independence means that applications need not be designed for particular database designs. Thus, as the logical database design changes, applications remain viable. At best, some tuning may have to be done, but not if the RDBMS supports a good statistical optimizer. Support for schema independence has a strong impact on maintenance costs. This means that applications do not need to know about database design particulars such as table and column names. Schema independence is usually achieved by using ~~with~~ stored procedures, rather than through embedded SQL. Care must be taken when evaluating stored procedure support, since not all stored procedures are created equal, nor are the languages in which they are written.

A great deal of application code can be eliminated if the RDBMS is capable of enforcing integrity rules, including entity, referential, domain, column, and user-defined integrity. Declarative primary key, column constraints, and referential integrity support, along with database triggers, provide much of this capability. Unfortunately, most RDBMS product do not

supports relational domains. Further application coding can be reduced is non-procedural access is well supported. The RDBMS must have a good optimizer and a good concurrency control scheme or else developers will write procedural code to circumvent RDBMS inefficiencies. have to

B. Operational and Maintenance Costs

Operational costs and resource utilization are important from the point-of-view of costs of ownership. Backup, restore, physical organization, reorganization, and restart procedures should be easy to manage. Tuning should be repeatable, the impact of changes easy to understand, and the process of tuning easy to and perform. The RDBMS should provide utilities for monitoring and governing resources usage, since this has an impact on costs associated with availability, maintenance, performance, and capacity planning. In addition, the RDBMS should be frugal with respect to physical resource utilization. Obviously, this will reduce costs associated with hardware support of the RDBMS such as CPU, disk, and memory requirements.

C. Direct Costs

Direct costs associated with licenses and software support agreements are also important. Typically, these are related to the number of users which must be supported and should be competitive in the industry. It is important to take a long term and broad view of these costs; it the features and functionality are not competitive, additional development and operational costs will be incurred and it is more likely that license upgrades will be required.

VII. DBMSs and Open Systems

An *extensible* RDBMS is achieved in a variety of ways. With the appropriate facilities, this leads to support for open systems. Open systems support has different meanings for different companies. For some, it means that integration and interoperation with their products is possible. For others, it means that public standards are supported. For still others, it means that proprietary standards are followed and made available to customers.

A. Interoperating with External Systems

There are several types of database *gateways* offered by RDBMS vendors and may be either software only or require a separate hardware platform. Gateways may support either uni-directional or bi-directional data access between two types of systems. Perhaps the two most common types of gateway are the standard RDBMS gateway and the custom gateway.

Standard gateways are provided by the vendor and provide access

to foreign data. Typically, the data is managed by another vendor's RDBMS products. There is tremendous variability in the degree of transparency and compatibility which such gateways provide. Gateways may provide either direct connection from an application or access via the native RDBMS.

Custom gateways are generally supported through vendor-supplied gateway development kits. This requires that the RDBMS support server-to-server interoperation and that the interface mechanism be fully documented and supported. Custom gateways permit the end-user to obtain access to proprietary and non-database data sources. This capability is extremely important for enterprise RDBMS environments, since it enables integration of RDBMS-based applications with legacy applications at the data level.

In addition to providing access to data, custom gateways are sometimes used to write user-defined extensions to the RDBMS. For example, if the user needs to implement data encryption services, a custom gateway can be written which performs this service. Without this capability, users must write such services into application code, are dependent on the RDBMS vendor to provide such services, or else must have the ability to modify the RDBMS code. Each of these options makes enterprise IS susceptible to additional and less controllable points of failure.

B. Standards

Ideally, the goal is low-cost extensibility leading to systems with a longer life expectancy. As long as limited functionality and slower innovation can be tolerated, low-cost extensibility can be achieved best by support for public or *de jure standards*. In principal, competition between vendors can lead to lower cost and easily interconnected applications. When higher levels of functionality are required, users must look to proprietary standards and the innovations of a particular vendor. If the vendor is large enough, such proprietary standards may become public by default (*de facto standards*). Of course, either de jure or de facto standards may restrain the industry from creating truly innovative, more efficient, and more cost-effective solutions.

Perhaps the most important RDBMS de jure standard is the *ANSI SQL* standard. This standard provides a definition of the syntax of SQL and was first approved in 1986. It was extended in 1989 with the Integrity Enhancement Features. Many vendors support some degree of compliance with the 1989 standard. Although ANSI does not validate compliance with the standard, the National Institute of Standards and Technology (NIST) does perform compliance testing for a superset of the standard, the Federal Information Processing Standard 127.

A new version of the standard, SQL2, is expected to be approved in late 1992. This version provides significant extensions to

the standard SQL syntax and functionality. Although RDBMS vendors are rushing to support some level of compliance with SQL2, it is extremely complicated, and considerable effort will be required. Yet another version, SQL3, is already in preparation and threatens to be even more complex.

Among the most confusing of enabling database technologies is that for *distribution.* Part of confusion stems from the difference between process distribution (distributed computing) and data distribution (distributed database). It is important that an RDBMS product support both. An RDBMS can support distributing computing through its architecture and through certain features such as database stored procedures (either local or remote). The most common distributed computing architecture supported by RDBMS products is client/server. A client/server architecture permits application-specific code to be separated from the RDBMS code and data, possible across multiple client platforms. Support for remote stored procedures permits database processing to be shared across multiple server platforms, but not necessarily within the context of a single transaction nor with each server having access to the same data.

RDBMS distributed database support requires that each user or application perceive multiple databases as a single logical database, regardless of physical location. Database authorities have proposed different ways of defining distributed database support. Chris Date has identified twelve objectives for distributed database support. These include local autonomy, no reliance on a central site, continuous operation, location independence, fragmentation and replication independence, distributed query processing, distributed transaction management, hardware independence, operating system independence, network independence, and RDBMS independence. Similarly, Mike Stonebraker has identified seven rules based on transparency. These are update, retrieval, scheme, performance, transaction, copy, and tool transparency.

While there are no distributed RDBMS standards per se, distribution of SQL requests requires interconnection. This area is the topic of a number of standards efforts. Among them are ISO's RDA (International Standard Organization's Remote Data Access), SAG's (SQL Access Group's) SQLCLI, IBM's DRDA (Distributed Relational Database Architecture), and Microsoft's ODBC (Open Database Connectivity). RDA provides a set of data formats and protocols without specifying either SQL syntax or an API. These issues are addressed by SAG's SQLCLI, which is a subset of Microsoft's ODBC. For a variety of reasons, IBM has chosen not to support SQLCLI and have defined a proprietary standard which they hope will become a de facto standard in IBM environments.

When existing environments are transaction-based, new technology cannot be integrated without some means of coordinated transactions between applications. Because traditional RDBMS products manage transactions through internal services, it is

important to support external transaction services, known as *TP monitors*. Both UNIX-based and proprietary TP monitors are available. Most UNIX-based TP monitors support a call-level interface standard known as *XOPEN/XA*. If the RDBMS supports TP monitors, newer applications can be integrated with older applications via the TP monitor at the transaction level. This means that older application modules can be selectively and more readily replaced by newer software without disruption of the existing transaction profiles.

In addition to standards for SQL, interconnection, and transaction processing, operating systems standards have also been developed. Compliance with operating system standards are intended to increase the portability and scalability of applications, as well as providing interoperation among systems. Most dominant in this area are standards for some form of the UNIX operating system. These efforts are usually known as open systems standards, although this is probably a misnomer since it restricts the idea of open systems to UNIX. Among the standards are OSF/1 and POSIX. Even proprietary operating systems such as Digital Equipment Corporation's VMS are moving toward *de facto* standards compliance by introducing an OSF/1 and POSIX compliant layer.

VIII. Conclusions

Great care must be taken in evaluating RDBMS products. Contrary to the predictions of non-technical industry analysts, extensive experience in evaluating RDBMS products shows they are not becoming commodity items. On the one hand, the differences between these products is becoming more subtle and difficult to understand. On the other, these subtleties make the difference between success and failure. If all RDBMS products were identical, the cost of replacing one with the other would be limited to the license fees and DBA training costs.

In this paper I have examined the key concerns of enterprise IS and the requirements these concerns place on RDBMSs. Given the costs, enterprise IS cannot afford to select the wrong RDBMS.

References:

D. McGoverna, "ORACLE7", Database Product Evaluation Report Series, c. 1992, Alternative Technologies.

