



Which Database is Right for the New Information Systems?

The purpose of this document is to show how to build scalable information systems (IS) for highly scalable transaction-based business applications on the Internet, using best-of-breed technologies to deliver new functionality that encapsulates and leverages investment in existing data sources and services.

Contents

Contents	2
Figures	3
Which database is right for the new information systems?	4
The Problems with the Relational Model	4
Complex Associations	4
SQL	5
Architecture	5
The Object Model	6
Modern Object Models	6
Object Interfaces	6
Other Database Options	7
Using objects in relational databases	7
Object-Relational (Hybrid) Systems	8
Mapping Tools	9
Memory Databases	10
The Advantages of Pure Object Databases	10
Storing an Object Model	10
Object ID	11
Navigation	11
Transparency	12
Distribution	12
Object Query	12
Object Cache	13
The Objections to Objects	13
Administration Issues	13
Training Needs	14
Reliability Worries	15
Scalability Concerns	15
Application-Dependent Information	15
Usability for Batch Processes	16
Data Migration	16
Conclusion	18
About Versant	19
Versant Overview	19
Versant History, <i>Innovating for Excellence</i>	19

Figures

Figure 1 - Complex Associations5
Figure 2 - Storing an Object Model10
Figure 3 - Object ID11
Figure 4 - Training Needs14
Figure 5 - Data Migration17

Which Database is Right for the New Information Systems?

The purpose of this document is to show how to build scalable information systems (IS) for highly scalable transaction-based business applications on the Internet, using best-of-breed technologies to deliver new functionality that encapsulates and leverages investment in existing data sources and services.

While these new systems will rely on object technologies, they also will use existing, traditional (non-object) data.

With this in mind, the question is, *Which database (DB) should be used for the new IS?* This DB will be used for meta-data and process repositories, for caching, for new complex components, and more - bringing into question whether traditional relational database management systems (RDBMS) are suitable for these new needs.

The Problems with the Relational Model

Relational databases are basically a set of tables, which are arrays of cells. Columns define the table structure, rows contains data. And that's it.

This means you will be forced to think and describe the world in terms of arrays, and design your data model with these simplistic concepts.

To be able to build complex data tables, you need to assign special meaning to some columns. For instance, one set of columns will allow you to uniquely identify each row in table A. This is then the primary key.

Complex Associations

This primary key must be duplicated in another table B in order to rebuild complex information. These columns will have a referential integrity constraints on A's primary key. Columns duplicated in B have a foreign key on A's primary key.

When you compose a B row with an A row, using the common values of the primary and foreign keys, you perform a *join* operation. If no indices are defined on A's primary key, this operation has an exponential cost, since the engine needs to scan all rows in A. But even with indices defined, this operation may explode if too many tables are joined (typically more than 10). Any relational programmer knows that when there are too many join clauses in the *where* part of an SQL statement, the order in which these clauses are written will have a huge impact on response time.

The only way to join rows from different tables is to manually duplicate values in key columns. Though you can join rows on columns not included in keys, this should mean that the data models are independent of the way data will be used. However, a join on non-key columns is just senseless! This is also true for joins based on operators other than *equal*.

Let's illustrate this with a simple example:

Invoice	Customer
Number	Id
Date	Address
CustomerId	Discount
Amount	

Figure 1 - Complex Associations

You can write statements like:

- `SELECT * FROM Invoice A, Customer B WHERE A.Number = B.Id`
- `SELECT * FROM Invoice A, Customer B WHERE A.Customer.Id > B.Discount`

No doubt this is a powerful statement, but what does it mean, and what is the cost? The fact is that 99.99 % of joins are based on keyed columns, which clearly indicates a weakness of RDBMS.

SQL

Another supposed strength of relational systems is that they all support a *normalised* declarative query language: SQL. This language has been designed to deal with the relational model. It allows to insert, update and delete rows (Data Manipulation Language); and enables programmers to add and drop columns, tables and indices (Data Definition Language).

The fact is that SQL is both too complex for simple users and too limited for expert programmers. Also SQL portability is less than advertised: even queries on dates are different on every implementation. Every relational vendor is compelled to define SQL extensions to make the system usable. Obviously, none of these extensions are normalised. Examples of such extensions are hints, recursive queries, outer joins, and so on.

From the programmers point of view, the issue with SQL is that this declarative language is not very well integrated into procedural programming languages.

Architecture

The last point about RDBMS is their massively centralised architecture. All database operations are performed on the server side, clearly creating bottlenecks, and performance degradations when simultaneous connections are increasing. Many RDBMS vendors have found more or less elegant solutions to this issue ranging from stored procedures to TP monitors.

The Object Model

Object models are richer than relational models. They have been used first in new programming languages, and later in modelling tools (uppercases).

In most of the current implementations, an object is an instance of a class. A class serves both as a model for object creation and as a set of all its instances.

Classes define the behaviour and state of their objects. State is defined by the values of a set of attributes. Behaviour is defined by a set of methods, or functions in traditional programming.

Modern Object Models

Modern object models define data and functions together. From that basis, we can add some properties:

- Encapsulation: the state of an object may be modified only by methods of this object
- Inheritance: a class may be defined as a specialisation of another existing class. It then inherits the state and behaviour of its parent class (its superclass). It can also add new methods or modify existing ones
- Association: an object can refer to others using references to them
- Polymorphism: different subclasses can implement the same method; the correct method will be called depending on the actual class
- Messages: objects communicate through messages.

In most object languages, attributes and methods may be private, public or protected. But usually, in a correct object design, every attribute should be protected in order to enforce encapsulation.

Public methods define the interface of a class (its protocol, or the set of methods that other objects can call). In other words, an object publishes several interfaces. Other methods are invisible and reserved for internal implementation.

Object Interfaces

So, the only way to manipulate an object is to use its interfaces. This will dramatically increase the quality of your system:

- Objects are protected and isolated. No other programmer can modify an attribute in an inconsistent fashion, therefore objects are robust
- Objects imply modular design at the deepest level, therefore objects are reliable
- Objects are self-testable: if the state is wrong, you only have to inspect the object's methods, making objects easy to maintain
- It's easy to plug a new object into the system, as all the interactions are properly defined in the interfaces, so objects systems can evolve quickly

-
- You can define knowledge at the highest level, and all subclasses will use it without redefinition, so objects are efficient
 - Polymorphism avoids the need to write *switch* control structures (if ... else if ... else if ...). As method names are dynamically bound, it's very easy to add a new type in the system without modifying existing control code, this means objects systems can evolve quickly,

All these reasons explain why object languages are widely accepted by the development community.

But the fact is, objects are mainly used for technical designs, and most of them are graphical interface objects, such as windows. Very few developers use business objects because business objects are persistent in most cases, and it's not easy to store objects in a traditional, relational database.

Other Database Options

Using objects in relational databases

Let's consider a very simple object model:

- Companies have a name and a set of customers
- Customers are either a company or a person
- A person is a special customer with an age
- Companies have departments
- Departments have a name and a set of employees
- An employee is a special person with a salary

It's very easy to draw this model using any UML tool and generate the associated Java source code. But issues come up when you want to store these Java objects into a relational database, because:

- You cannot express inheritance at all
- You cannot easily represent and use references between objects

So, as a programmer, you will have to figure out how to store and retrieve rich objects in a poor storage model—for example, using two-dimensional arrays, like spreadsheets.

Basically three different approaches may be used to deal with the inheritance issue:

- You can decide to store each class in a table,
- You can decide to store each class hierarchy in a single table,
- You can decide to store each hierarchical level in a separate table and manage the link with the upper level.

None of these approaches are perfect: all have tradeoffs.

With solution one, SELECT statements become quite cumbersome. This is because when you select Person objects, you have also to select Employee objects, which are classified as Person objects. So you have to call two SELECTS and then build the final result. Each time you add a new subclass, you must add a new SELECT to the list before computing the final result.

In some cases you can automate this complex mechanism. If your object language supports meta-classes you can use introspection to see if a class has subclasses, and you can program a *recursive select*.

However, this means you will not be able to easily express referential integrity constraints anymore. Even simple ones like *unique* are difficult because you have to check uniqueness among a set of tables, and this is not a standard feature of RDBMS.

With solution two, you compute a set of all the attributes defined in a class hierarchy. Then you create a table with all these attributes, plus a new one, the class name to which each row belongs. If equal attribute names are defined in several subclasses, it becomes a little bit harder to manage. The SELECT statement has exactly the same limitations as in solution one, unless you use a smart, or hierarchical class name. Either way, you'll waste a lot of disk space.

With solution three, each time you INSERT a new Employee, you will also INSERT a new Person, and you will have a foreign key defined on the Employee to link it to the corresponding Person. This means that each time you update your class hierarchy, you will have to change your code. Also, all the primary keys have to be duplicated in subclasses, which simplifies SELECT statements but wastes disk space.

There are also navigation problems. You can't store references (single pointers or dynamic collections of pointers) directly into a relational database. You also can't express foreign keys in object languages. This means you are obliged to rebuild your object references from foreign keys—a difficult job, even when the graph of the objects is not too deep. With more complex foreign keys, it becomes quite difficult.

Object-Relational (Hybrid) Systems

Object Relational Systems basically provide two new features:

- New data types
- Complex attributes

The first feature allows new systems to deal with new kind of data such as multimedia or time-series, spatial co-ordinates and more. These new data types are more or less integrated with the underlying relational structure and query language. Moreover, they are not object-oriented since there is no support for inheritance. However, though hybrid systems allow you to deal with new, more complex data *types*, they don't offer anything to cope with new, complex *data models*.

The second feature allows you to create a new table with another table as an attribute. This is supposed to simulate an association in an object model. It's important to notice that this is only a simulation at the syntax level. From the storage point of view, there are still two separate tables, and information retrieval still uses time-consuming joins.

This is a first step toward object systems. But it is also probably the last one, as other object features such as inheritance, are much more difficult to simulate with RDBMS.

Mapping Tools

Mapping tools allow programmers to associate rows in relational tables to instances of classes (i.e. objects). They also propose to dynamically view rows as pure objects, replacing foreign keys with real pointers.

This makes them a very interesting way to use existing relational data in new object applications. When used to map existing data models, they are very efficient, because existing relational models are very simple by definition.

However, some issues remain:

- Mapping a complex object model is a very long task, and these products offer very poor assistance. For each attribute of your object model, you have to manually indicate which table columns you will store it to, or retrieve it from
- They generally support different mapping algorithms, all of them implying that you store the class name with the row—not a particularly elegant design solution
- Pointers and collections are mapped to columns. Mapping object IDs to primary and foreign keys, generates corresponding internal values. This locks your data into hard-coded, black-box mechanisms. As soon as there's a bug, these kinds of systems become very hard to maintain
- The foreign key mapping wastes a huge amount of disk space. This also happens if you choose a mapping strategy where a whole class hierarchy is stored in a single table
- These tools cannot eliminate mistakes during the mapping phases. So, if you associate the CustomerName attribute to the CustomerID, you won't be notified, and it may take months before you discover the disaster
- Even if you don't have to write the mapping code yourself, this code exists and will be executed, having a huge impact on global, visible response time
- Because of poor performance, these systems propose a lot of optimizations and hints. All these tuning strategies will directly impact your object design. For instance, a whole object graph is completely loaded by default when you read an object. This is mandatory to simulate transparent navigation. But it is also not scalable. Mapping tools do propose a means to manage loading granularity, but to do that you must use a special *ValueHolder* instead of a simple object pointer. So you have to change your business model to suit their technology!

-
- Even with all these tricks, mapping tools don't provide transparent navigation into the object model. The code you write is actually linked to the mapping tool and sometimes to the mapping algorithm you chose.

Memory Databases

Memory databases are transactional, in-memory, relational engines that asynchronously replicate themselves into a standard (on-disk) RDBMS. But, Memory databases still suffer many problems such as: no object support, physical memory limitations, no support for distribution, not shareable between several Enterprise Java Bean (EJB) server instances, no smart policies to manage synchronisation between memory cache and physical volumes, no support for triggers.

The Advantages of Pure Object Databases

Object Database Management Systems (ODBMS) offer a new paradigm to fully support objects in the persistence layer of an information system. This section describes common features of any ODBMS, as standardised by the Object Data Management Group (ODMG).

Storing an Object Model

The basic idea of an ODBMS is to *directly* store any complex object model into a database, with no limitations of the object-oriented concepts supported. The ODBMS is not a syntactical representation wrapped around an RDBMS, but really provides full database management with object physical storage.

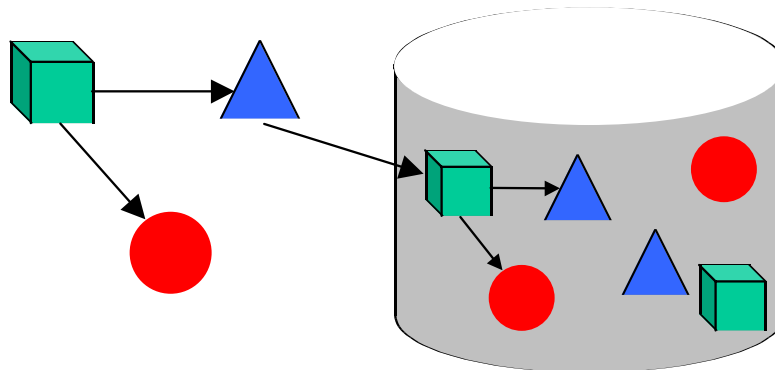


Figure 2 - Storing an Object Model

Object ID

Each time a new object is stored into the database, the engine will assign it a unique Object Identifier (OID). Depending on the implementation, this identifier may be physical or logical, and distributed or not.

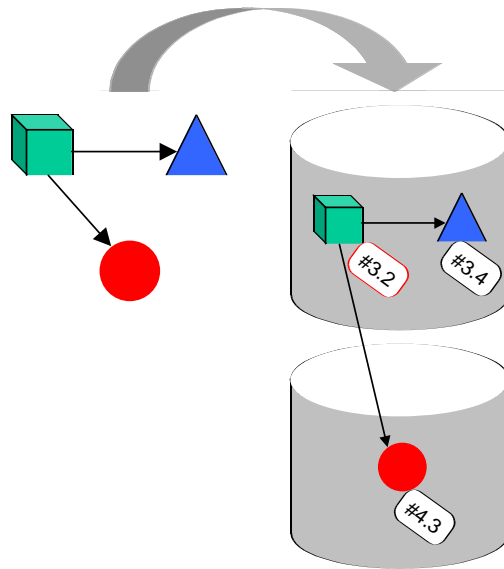


Figure 3 - Object ID

Navigation

One of the strengths of the object model is the ability to define references, or relationships between objects. Using object languages, these references are implemented with pointers. When storing an object with references to other objects, the ODBMS will transform volatile, in-memory pointers into smart, persistent ones. To do that, OIDs are used as persistent, unique references. So, once in the database, objects refer to one another through OIDs. Once reloaded into memory, OIDs are transparently mapped into pointers.

Navigating with OID is not only a more elegant, but also a very efficient way to retrieve objects.

Let's go back to our previous example with customers and invoices.

Suppose you have one invoice in memory, and you need to access its customer. Using SQL, you will write something like:

```
SELECT * FROM Invoice, Customer WHERE Invoice.CustomerId = Customer.Id
```

This string will be sent from the client to the RDBMS server. There, after syntactical parsing and checking, a tree search is built. The optimizer will try to improve the performance. Then, the query is executed and the result set is built and sent back to the client. The client will iterate through this result set, and create Java objects for each of its rows.

Let's do the same with an ODBMS, in pure Java:

```
oneInvoice.customer;
```

Simply using one attribute (which in this case is a reference to a customer) in a simple Java statement, the customer will be found in the database and one Java Customer object instantiated in memory. At the server side, there is no need for syntax checking or optimization, since the customer attribute has been stored within its Invoice object in the form of the customer's OID. Retrieving objects by navigation may be up to 100 times faster than using queries—even with proper indices defined in the RDBMS! At the client side, there is no need to iterate and to recreate objects from a raw, flat, array-like result set of rows.

Transparency

Transparency is a way to store and retrieve objects in a database without explicit, declarative query statements. Transparency relies on OID and additional built-in mechanisms. Each time you use a reference on an object, if that object is not already in memory it will be automatically fetched from the server. On the other hand, each time you modify an object's state, it will be updated in the database at the next commit, without any need to paraphrase the Java code with database code.

```
oneInvoice.customer.name = Versant;  
  
// will load the customer into memory on-the-fly if not already there,  
  
// and will update it at the next commit into the database
```

Distribution

Some ODBMS support distributed storage and retrieval. Both horizontal (every class is defined in each database) and vertical (some classes in one database, other classes in other databases) partitioning may be supported. Distributed storage means that you can use specific policies such as Round Robin, to store objects in different physical databases. Distributed retrieval means that parallel queries are supported.

Native distribution is a key feature for load balancing purposes.

Object Query

Some ODBMS come with a query language that allows you to deal with object concepts like inheritance, navigation and distribution.

For example, with Versant Query Language (VQL) you can write:

```
SELECT * FROM ONLY Company WHERE departments->manager->name LIKE 'Eric*' IN ALL
```

The ONLY modifier means you don't want instances of Company subclasses to be included in the result set.

The -> means that you traverse object references (including collections of references) during the query. In this example, *departments* is a Java vector attribute in Company, *manager* is a reference to a Manager object in the Department class, and *name* is a simple string attribute in Manager class.

The IN ALL clause states that this query will be executed in parallel on any connected database, with one simple result set built at the client side.

Object Cache

When objects are loaded into memory; either implicitly, during transparent navigation, or explicitly during queries; they are in a special area of the Java Virtual Machine heap, under the control of an object manager. The OM will translate pointers to OID and back, will transparently fetch objects from the server whenever necessary, and will mark modified objects to be flushed to the server at the next commit.

When an object has already been fetched during a transaction, the next time it is referenced it will be located very quickly in the client cache. By default, this efficient client cache is cleaned at the end of each transaction, but more sophisticated mechanisms are also available.

The Objections to Objects

ODBMS are so efficient and elegant that you may now wonder why they are not yet widely accepted by the IT industry.

Administration Issues

While ODBMS vendors claim easy administration, RDBMS vendors usually claim the opposite. The truth is somewhere in between.

ODBMS are faster, so no tuning is required. However, because ODBMS are often used in environments where speed constraints are much higher, optimization will still be needed. Management of security, user access, backups, disk occupation, available memory, hardware usage, OS configuration — daily database administration tasks, remain exactly the same, whatever database technology you may use.

Most existing ODBMS engines do not allow for online administration, and their software architecture is really different from a traditional DBMS. You could actually consider them more a persistent language with access to shared disk resources, rather than a real database engine. These are the real reasons so many administrators are reluctant to use ODBMS.

Versant technology has been designed by senior architects from the database world. So Versant ODBMS offer:

- Server software architecture which is quite similar to an RDBMS. For example, there is a listener daemon waiting on a dedicated port, a multi-threaded process accessing shared memory segments, and more.

- Most database administration tasks may be performed online
- Database administrators can use standard SQL to update objects

Training Needs

Many IS directors reject ODBMS because they think they require extensive initial training to master the technology. In actuality, what you have to learn is Java. And the learning curve is more than balanced by the increased return on investment.

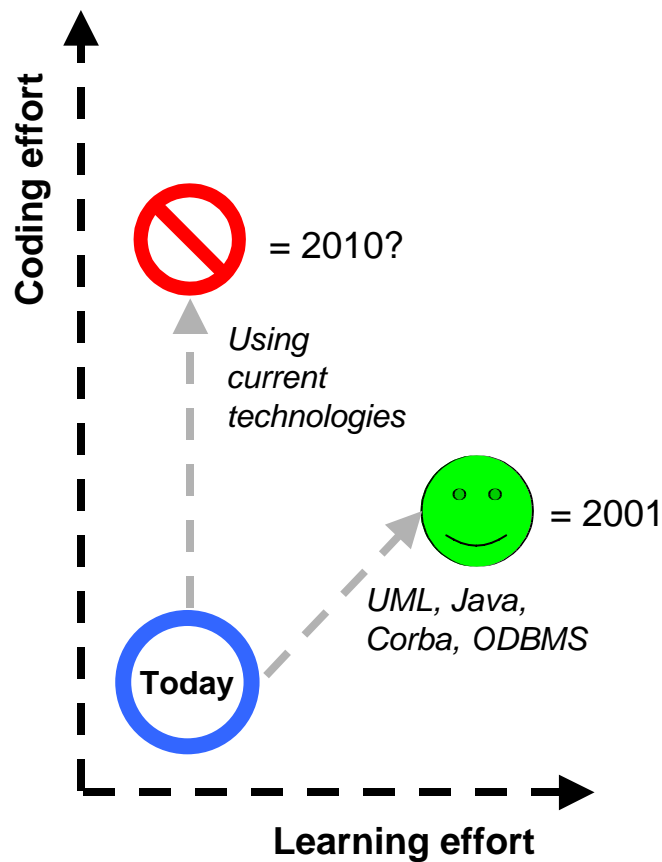


Figure 4 - Training Needs

Let's look at a typical RDBMS Java code sequence:

- Select something from database
- Iterate through result set, then recreate objects
- Do something to your objects using Java

-
- Reflect updates in the database
 - Commit changes to the database

Using Versant it's exactly the same, you just have to skip steps 1, 2 and 4!

Reliability Worries

It is true that many ODBMS implementations are not really usable in high-availability environments. However, Versant technology can because:

- Versant provides a very sophisticated logging system, with automatic crash recovery on start
- Versant provides online distributed backup at several levels (i.e. one full backup per month, one sub-backup per week, one sub-backup per day)
- Versant provides automatic log archiving to prevent any loss of data after a disk crash
- Versant is the only ODBMS vendor which provides a genuine out-of-the-box fault tolerant server; enabling online restore
- Versant has a list of large projects deployed in 24x7x52 environments

Scalability Concerns

It is generally well recognised that ODBMS are faster than RDBMS because of the navigation support through direct object references (OID). But speed is not scalability.

The fact is that most of the existing ODBMS implementations have a very poor network layer for data transport. This leads to dramatic increase in response time as soon as concurrent transactions or database size increase, just because the network cannot accept the overhead.

The Versant engine works like a traditional RDBMS in functions like query processing and index management. Because of this, and because Versant has a true *object server* architecture—versus a *page-based* design that works like OS virtual memory mechanisms—Versant ODBMS can scale. Versant is not a persistent language but a real database engine dealing with objects. Objects are stored independently (not in pages); so queries, indices, network traffic and locks are fully optimized.

Also, the Versant engine is a highly multi-threaded process that uses multiple latches (DBMS critical sections), allowing you to really take advantage of SMP architectures.

Application-Dependent Information

It's widely accepted that ODBMS tend to dramatically reduce development cycles. It's also well recognised that persistent smart pointers greatly improve performance. However, this raises concerns that common, most frequent access paths, are hard-coded into the database, so that data may be very difficult to read outside applications—for instance, using ad-hoc queries.

Let's try to show this, using our Invoice and Customer classes.

If you model this in Java with a Customer reference attribute in the Invoice class, it will be very easy and efficient to get the customer associated to one invoice:

But, conversely, it becomes quite impossible to quickly retrieve all the invoices of a given customer.

```
Invoice oneInvoice = Invoice.selectByInvoiceNumber(aNumber);  
Customer oneCustomer = oneInvoice.customer; // get the customer of this invoice
```

The solution here, is bi-directional references. With bi-directional references you store cross-references at both sides of the relationship between two classes. If you provide the code to manage this association, your object model becomes as open as a well-normalised relational one.

Unfortunately, this kind of pattern is neither supported by Java or by UML code generators.

Usability for Batch Processes

We have seen that ODBMS are very fast. We have also seen that Versant Query Language is scalable because it has been designed like RDBMS query engines.

But, to be honest and complete, we should recognise that the current implementation of VQL only supports scalable query shipping for SELECT statements, and does not support projections - retrieving only some attributes from objects. Also VQL does not support method execution in WHERE clauses.

This may be a real issue in batch-oriented applications.

Data Migration

When large enterprises consider ODBMS, they often face the problem of how to integrate them with existing data.

Existing applications generally produce text files that must be integrated in other applications. Even RDBMS export data in CSV files. This is because the tabular nature of CSV files perfectly matches the non-complex data format of older applications.

But CSV files are useless to integrate data into ODBMS as they cannot represent object references or Java collections.

This is why Versant now provides a dbLoader, that allows you to load simple data formats such as CSV files into a real object model. In order to do that, the dbLoader uses an external XML file, which contains the necessary meta-data to *objectify* raw text-like data.

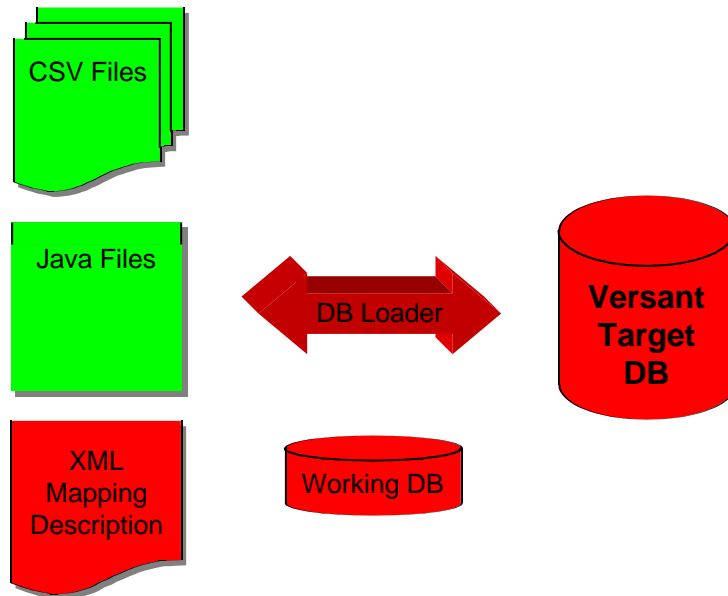


Figure 5 - Data Migration

Versant also provides XML import/export facilities, using a canonical Versant DTD. These XML files may be translated into other DTD using XSL-T.

Conclusion

Existing legacy systems are already overloaded, and cannot easily evolve.

Many business processes are migrating to the Internet. However, moving applications to the Internet multiplies constraints on existing systems by a factor of ten.

This scalability issue cannot be solved by out-of-the box hardware or software technologies. Scalability is more a question of culture, organisation, and architecture.

Object technologies, mainly Java, will allow companies to cope with new functional and evolutionary constraints. But EJB servers, as currently standardised, won't be able to tackle the scalability issue alone. EJB standards must evolve to take full benefit of object technologies, including ODBMS-oriented containers, used either as smart caches or new data repositories.

About Versant

Versant Overview

As a leading provider of object-oriented middleware infrastructure, Versant offers the **Versant Developer Suite** and **Versant enJin**. Versant has demonstrated leadership in offering object-oriented solutions for some of the most demanding and complex applications in the world. Today, Versant solutions support more than 650 customers including AT&T, Alcatel, BNP/Paribas, British Airways, Chicago Stock Exchange, Department of Defense, Ericsson, ING Barings, MCI/Worldcom, Neustar, SIAC, Siemens, TRW, Telstra, among others. The Versant Developer Suite, an object database, helps large-scale enterprise-level customers build and manage highly distributed and complex C++ or Java applications. Its newest e-business product suite, Versant enJin, works with the leading application servers to accelerate Internet transactions.

Versant History, *Innovating for Excellence*

In 1988, Versant's visionaries began building solutions based on a highly scalable and distributed object-oriented architecture and a patented caching algorithm that proved to be prescient. Versant's initial flagship product, the Versant Object Database Management System (ODBMS), was viewed by the industry as the one true enterprise-scalable object database. Leading telecommunications, financial services, defense and transportation companies have all depended on Versant to solve some of the most complex data management applications in the world. Applications such as fraud detection, risk analysis, yield management and real-time data collection and analysis have benefited from Versant's unique object-oriented architecture.

VERSANT Corporation

Worldwide Headquarters

6539 Dumbarton Circle
Fremont, CA 94555 USA
Main: +1 510 789 1500
Fax: +1 510 789 1515

VERSANT Ltd.

European Headquarters

Unit 4.2 Intec Business Park
Wade Road, Basingstoke
Hampshire, RG24 8NE UK
Main: +44 (0) 1256 366500
Fax: +44 (0) 1256 366555

VERSANT GmbH

Germany

Arabellastrasse 4
D-81925 München, Germany
Main: +49-89-920078-0
Fax: +49-89-920078-44

VERSANT S.A.R.L.

France

10 rue Troyon
F-92316 Sèvres Cedex, France
Main: +33 1 450767 00
Fax: +33 1 450767 01

VERSANT Italia S.r.l.

Italy

Via C. Colombo, 163
00147 Roma, Italy
Main: +39 06 5185 0800
Fax: +39 06 5185 0855

VERSANT Israel Ltd.

Israel

Haouman St., 9
POB 52210
91521 Jerusalem, Israel
Main: +972 2 679 38 30
Fax: +972 2 679 61 49

VERSANT India Pvt Ltd.

India

1240-A, Subhadra Bhavan,
Apte Road, Shivajinagar
Pune 411 004, India
Main: +91 20 553 9909
Fax: +91 20 553 9908



1-800-VERSANT
www.versant.com

WP_001001r6
© Versant Corporation 2000
Reprinted in 2001

All products are trademarks or registered trademarks of their respective companies in the United States and other countries.

The information contained in this document is a summary only.

For more information about Versant Corporation and its products and services, please contact Versant Worldwide or European Headquarters.