Lessons from Wall Street: case studies in configuration, tuning, and distribution

Dennis Shasha
Courant Institute of Mathematical Sciences
Department of Computer Science
New York University
shasha@cs.nyu.edu
http://cs.nyu.edu/cs/faculty/shasha/index.html

Abstract

Consider a setting in which

- Database speed and reliability can make the difference between prosperity and ruin.
- Money for information systems is no object.
- Data must be accessible from many points on the globe with subsecond response.

The financial industry is exactly such an environment.

This tutorial presents case studies in configuration, tuning, and distribution drawn from financial applications. The cases suggest both research and product issues and so should be of interest to the entire Sigmod community.

1 Preparing for Disaster

If you take a database course in school, you will learn a model of recovery that features a convenient fiction: stable storage. The idea is that failures affect only volatile storage, whereas stable storage survives failures unscathed. In some lecture halls, disks soon acquires the character of stable storage. Discussions that admit the possibility of disk failure posit the use of a disk mirror or a RAID disk to do away with this problem. Do you believe this fiction?

Case: Server for trading bond futures having to do with home mortgages.

The application needs to be up only a few days per month, but the load is heavy during those few days. During a weekend batch run, 11 out of 12 2 gigabyte disks from a single batch from a single vendor failed. Nothing terrible happened, but everyone was quite spooked.

The system architect didn't want a backup machine because his understanding was that the backup could catch up to the primary only after 20 minutes. His traders couldn't stop trading for that long, so would have reverted to calculations by hand at which point they would be unwilling to reenter trades to the system.

In this organization, policy requires that backups be remote, because an entire building may suffer from a fire or bomb scare. (This happens in real life. Credit Lyonnais's Paris office burned up a few years ago. Smoke has shut down the New York stock exchange's principal computers. We won't even talk about the World Trade Center. In both cases, remote backups allowed continued operation.)

So, as a first case study, how would you design a remote backup? Here are some of the high availability choices typically made on Wall Street.

- **shared disk high availability servers** — A pair of shared memory multiprocessors attached to RAID disks. If the primary multiprocessor fails, the backup does a warm start from the disks. If a disk fails, the RAID parity prevents any error from being visible (and the faulty disk is replaced by a hot spare). This configuration survives any single disk plus any single processor failure, but does not survive site disasters or correlated failures. So, it is not a remote backup solution.
- **dump and load** — A full dump of the database state is made to a backup site nightly and then incremental dumps (consisting of the changed page images) are made at regular intervals (e.g., every 10 minutes) during the day. If the primary site fails, the backup can perform a warm start from the last incremental dump, but committed transactions since the last incremental dump will be lost. Dump and load imposes little overhead on the primary server, but works only when one can assume there is some redundant record of recent committed transactions. It also assumes the traders will be patient during the several minute delay before the system comes up, not always a likely assumption.
- **replication server** — As in dump and load, a full dump of the database state is made to the backup site nightly and then all operations done to the primary are sent to the secondary (typically, as SQL operations) after they are committed on the primary. With replication, the backup is within a few seconds of being up-to-date. So, replication loses fewer transactions than dump and load. Further, replication allows decision support queries to the backup machine. Finally, the backup machine can become the primary in a matter of seconds. Replication can be a huge administrative headache however, requiring up to 25% of the time of an administrator to keep table schemas in synchrony, to ensure that triggers are disabled at the backup site, and to be ready when the network connection breaks.

- **remote mirroring** — Writes to local disks are mirrored to disks on a remote site. The commit at the local machine is delayed until the remote disks respond. This ensures that the remote site is up-to-date, but may stop the primary in case the remote disks do not respond or the network link fails. To overcome this availability problem, special subsystems or TP monitors will buffer the transmission from the local to the remote in which case the result is similar to replication server.

- **two phase commit** — Commits are coordinated between the primary and backup. If one fails, blocking can occur. This fact scares off many people on Wall Street.

- **wide-area quorum approaches** — DEC, Hewlett Packard, IBM, ISIS, and others offer some subset of the following architecture (DEC, for example, offers all). Primary and secondary servers are interconnected via a highly redundant wide area cable. Clients can be connected to any server since the servers execute concurrently with their interaction managed by a distributed lock manager. Disks are connected with the servers at several points and to one another by a second wide area link. Heartbeats monitor the connectivity among the various disks and processors. If a break is detected, one partition holding a majority of votes continue to execute. Other partitions don't. In this architecture, backups are completely up-to-date. Any single failure of a site, processor, or disk is invisible to the end users (except for a loss in performance). The same holds for many multiple failure scenarios. According to DEC, most money transfer applications and most major exchanges use this architecture.

As far as I can tell, the wide-area quorum approach is the most promising as in Figure 1.

People who don't use this architecture site the following reasons:
(i) the shared lock manager is a potential bottleneck.
(ii) partitioning may occur improperly.
(iii) it locks them into a proprietary and expensive architecture.
(iv) specialized software priests must minister to its needs.

In practice (iii) and (iv) dominate the considerations in organizations I have seen. In the context of this case study, dump and load is the current solution and replication will be the eventual solution.

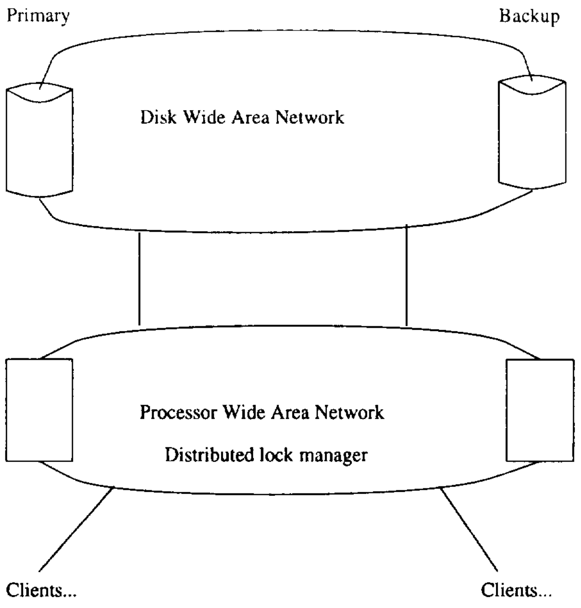
2 When Relational Systems Fall on their Faces

Case: You must price a bond based on future payment schedules and interest rate speculations.

The data comes in the form of arrays where the ith position corresponds to the ith payment. Financial calculations deal heavily with declining exponentials. Relational systems are thought to be way too slow for these array style calculations.

Product Question: Isn't it time this array problem were overcome? In other words, how much longer must I dissuade people from using Excel as their database manager? Are object-relational systems in fact fast enough?

A standard workaround is to attach a trigger to an insert of a set of bond positions (positions are trader, bondid,



Quorum Approach as Used in most Stock and Currency Exchanges.
Survives Processor, Disk, and Site failures.

Figure 1: Quorum approach used in most exchanges.

quantity triples). The trigger breaks down a collection update into a row-by-row calculation as follows.

```
for each trader-bondid pair
  do preliminary setup to bring in the
        calculation method and parameters
        for this bond
  depending on the bondid in the row,
        call the appropriate C calculation routine
end for
```

Suppose the trigger takes a long time to execute. The problem may be the C routine. Since the calculation depends only on on the bond rather than on the trader-bond, some time can be saved by performing calculations on one bond at a time.

But in my experience the problem usually is that the preliminary setup takes too long. This step is commonly a foreign key join that brings in information. Doing it on a bond-trader pair basis is very inefficient. Moving the join outside the loop is much better, 8 times better in one experience I had.

Case: Indicative Data Display

Lookup information such as symbol-name mappings must be at a trader's fingertips. Relational systems or the connections to users' personal computers are often too slow, so the information is held outside the database.[1] The problem is that this information is updated from time to time (e.g., when new securities or new customers are added)

So, a critical question is what to do with the updates. The following are all tried:

[1]These performance complaints are not unique to Wall Street. I understand that IBM data mining algorithms remove data from DB2 and run them from the file system.

- Ignore updates until the next day.
- Program clients to request refresh at certain intervals.
- Have the server hold the state of the clients. Send messages to each client when an update might invalidate an out-of-date copy or simply refresh the screens.

Which would you choose?

Product Question: Could you help somehow? That is, consider the general problem of functionality that belongs outside the database because it is too computationally or graphically intensive for a database application. How best should you keep the data used by that application up-to-date?

3 Interoperability with Other Databases

As in many industries, financial data must travel from clients to mid-office servers to a variety of back-office servers and risk management systems. Acknowledgements and corrections travel back.

Case: You have a front office system that records trade for profit and loss and risk management purposes. You want to send trades to the back office for final clearing. You don't want to lose any trades, because of the financial loss implied. How should you solve this problem?

Replication server is one approach, but is deemed too unreliable. One academic recommendation is to use two phase commit, but this is widely mistrusted on Wall Street because of horror stories about blocking. In addition, there is the serious problem that many database applications fail to provide an interface to one phase commit.

A typical solution is to implement an "interface table" on the source system side that holds information in the denormalized form required by the destination system.

The destination database reads a new tuple t from the interface table. After processing t, the destination database flags t as deletable or deletes t itself in the interface table. Such a scheme frequently runs into blocking problems, because both the source and the destination are scanning the interface table for update purposes. What would you do about this problem?

Here are some typical solutions:

- The destination process can mark a table other than the interface table to avoid update-update conflicts on the interface table.
- If the exact order of updates at the destination site is not important, then the primary site can use a clustering index based on a hash of a key. The destination process can then check one bucket or another of the hash on each of its polling steps to the interface table. This reduces the likelihood of conflict.

Case: Another typical scenario is that a single transaction in a master database system performs complete transactions on several other foreign databases. If the transaction on the master aborts, some of the transactions on the foreign databases may have committed. When the master transaction restarts, it should not redo already committed foreign database transactions.

Avoiding this requires a judicious use of "breadcrumb tables."

4 Global Systems

Information businesses have global inventories that are traded around the clock on different markets. (Markets have in fact converted most atom businesses into bit businesses to use Nicholas Negroponte's terminology. Oil for example is traded several times while a tanker makes a single voyage.) In finance, stocks and bonds are traded nearly 24 hours per day (there is a small window between the time New York closes and Tokyo opens).

Global trading of this sort suggests two solutions: have a centralized database that traders can access from anywhere in the world via a high-speed interconnect. This works well across the Atlantic, but is expensive across the Pacific. It also makes people feel psychologically uncomfortable, since they are dependent on a far-away data source.

Replication is delicate for concurrency control reasons however. If updates on all data can happen anywhere, then serialization errors can occur.

Example: At 11:59 AM, account X has a cash position of 10 million dollars at sites A and B (say in New York and Tokyo). Suppose that Site A increases the cash position of account X by 2 million dollars and site B increases it by 1 million dollars at 12 noon. The update at site A is translated to a delete and insert sent to site B and similarly for the update at site B one second later. Two seconds later, site A writes the value that B had written at noon (yielding 11 million dollars) and B writes the value that A had written at noon (yielding 12 million dollars). The values are inconsistent with one another and with the truth (13 million dollars). This is a classic "lost update" problem and is avoided in centralized systems that make use of serialized transactions.

Gray et al.[1] have examined the lost update and related problems in a replication server setting from a probabilistic point of view. The error probability increases superlinearly as the number of sites increase. In the idealized model of that paper a factor of 10 increase in sites leads to more than 1000 fold increase in the probability of error.

Sometimes, one can exploit knowledge of the application to avoid a problem. If one knows, for example, that all updates are additions and subtractions, one can send the update operation itself to all sites, and they will all eventually converge to 13 million dollars. Such a replication strategy must, unfortunately, be explicitly programmed by the user if the database management system sends delete-insert pairs to reflect each update.

Case: A trading group has traders in 8 locations accessing 6 Sybase servers. Access is 90% local. Trading data (trades made and positions in securities) are stored locally. Exchange rate data, however, is stored centrally in London. Rate data is read frequently but updated seldom (about 100 updates per day). Wide-area network performance is a big issue and slows things down a lot. 64 kilobit links between the sites (even not all possible pairs) costs $50,000 per month.

Here are the two problems:

- For traders outside of London, getting exchange rates is slow. It would be nice if exchange rates could be replicated. But consistency is required in at least the following senses: (i) if a trader in New York changes a rate and then runs a calculation, the calculation should reflect the new rate. This consistency requirement precludes a replication strategy which updates a primary in London and then does an asynchronous replication to all other sites, because the replication has an unpredictable delay. (ii) After a short time, all sites must agree on a new exchange rate. Before reading on, what would you do?

Here is one possibility. Synchronize the clocks at the different sites. Attach a timestamp to each update of an exchange rate. Put a database of exchange rates at each site. Each such database will be read/write. Each exchange rate in the databases will be associated with the timestamp of its latest update. An update will be accepted at a database if and only if the timestamp of the update is greater than the timestamp of the exchange rate in that database. This scheme will guarantee consistency requirement (i) and will cause all sites to settle eventually to the most recent update, satisfying (ii).

- Trade data is mostly local, but periodically traders collect baskets of securities from multiple sites. The quantity available of each security must be known with precision. Right now, retrieving this information is slow, largely because of WAN delays. The current implementation consists of an index that maps each security to its home database and retrieve necessary data from the home site.

  A possible improvement is to maintain a full copy of all data at all sites. Not all of this data will be up-to-date ("valid") at all times however. When a market closes, all its trades for the day will be sent to all other sites. When receiving these updates, a site will apply them to its local database and declare the securities concerned to be "valid." When a site s opens for trading, all other sites will declare securities belonging to s to be "invalid."

The last paragraph of the last bullet brought up a point that is seen frequently on Wall Street: the need to rotate the ownership of data over the course of the day, depending on which market is open. Suppose New York assumes it inherits ownership from London at 11 AM New York time. If the connection is down when London loses its ownership, then the last updates that London did might be lost.

5 Overview of the Tutorial

This short article has had space for only a subset of the cases from the tutorial. Other topics include more examples of tuning, the use of main memory databases, object layers, and many complaints about current products. You can obtain a full copy of my tutorial notes by sending me email or through my web page.

6 Biography of the Speaker

Dennis Shasha is a Professor of Computer Science at New York University's Courant Institute of Mathematical Sciences. His three principle research projects concern
i. the design and implementation of pattern matching and discovery algorithms with applications to biology.
ii. the Thinksheet system, a unification of hypermedia, expert systems, and spreadsheets for complex information presentation.
iii. and parallel support for data mining on networks of workstations.

He also consults for some Wall Street companies on database tuning and configuration issues and for Bell Labs and Bell Communications Research on a variety of database, workflow, and scheduling issues.

Shasha received his BS from Yale in 1977. He then worked for three years at IBM where he designed circuits and microcode for the IBM 3090 while obtaining a masters from Syracuse University. In 1984, he received a Ph.D in applied mathematics from Harvard and came to NYU.

He has published four books: *Database Tuning: a principled approach* published by Prentice-Hall in 1992; a book of biographies *Out of their Minds: the lives and discoveries of 15 great computer scientists* published by Springer-Verlag in 1995; and two books about a mathematical detective named entitled *The Puzzling Adventures of Dr. Ecco* and *Codes, Puzzles, and Conspiracy*, both published by W. H. Freeman in 1988 and 1992 respectively. Like Dr. Ecco, he loves puzzles.

References

[1] Jim Gray, Pat Helland, Patrick O'Neil, and Dennis Shasha "The Dangers of Replication and a Solution" ACM Sigmod 1996, pp. 173-182.