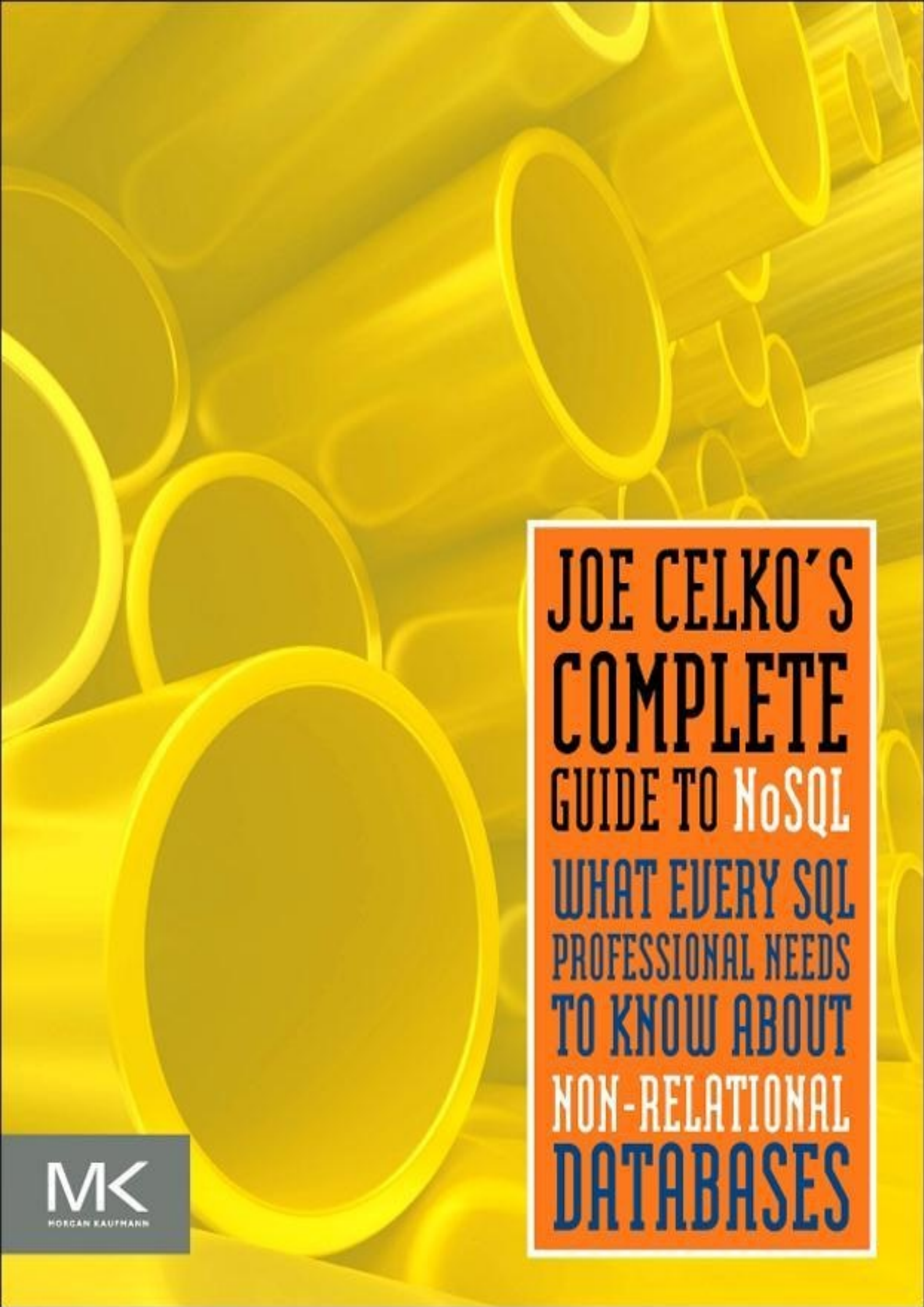

JOE CELKO'S
COMPLETE
GUIDE TO NoSQL
WHAT EVERY SQL
PROFESSIONAL NEEDS
TO KNOW ABOUT
NON-RELATIONAL
DATABASES
MK
MORGAN KAUFMANN

Joe Celko's Complete Guide to NoSQL

What Every SQL Professional Needs to Know about NonRelational Databases

Joe Celko

AMSTERDAM • BOSTON • HEIDELBERG • LONDON
NEW YORK • OXFORD • PARIS • SAN DIEGO
SAN FRANCISCO • SINGAPORE • SYDNEY • TOKYO
Morgan Kaufmann is an imprint of Elsevier

Table of Contents

Copyright

Acquiring Editor: Andrea Dierna

Development Editor: Heather Scherer

Project Manager: Punithavathy Govindaradjane

Designer: Mark Rogers

Morgan Kaufmann is an imprint of Elsevier

225 Wyman Street, Waltham, MA 02451, USA

Notices

Knowledge and best practice in this field are constantly changing. As new research and experience broaden our understanding, changes in research methods or professional practices, may become necessary. Practitioners and researchers must always rely on their own experience and knowledge in evaluating and using any information or methods described herein. In using such information or methods they should be mindful of their own safety and the safety of others, including parties for whom they have a professional responsibility.

Library of Congress Cataloging-in-Publication Data

Celko, Joe.

Joe Celko's complete guide to NoSQL : what every SQL professional needs to know about nonrelational databases / Joe Celko.

pages cm

Includes bibliographical references and index.

ISBN 978-0-12-407192-6 (alk. paper)

1. Non-relational databases. 2. NoSQL. 3. SQL (Computer program language) I. Title. II. Title: Complete guide to NoSQL.

QA76.9.D32C44 2014

005.75–dc23

2013028696

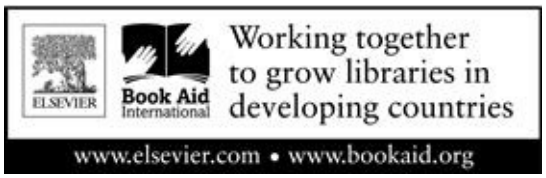
British Library Cataloguing-in-Publication Data

A catalogue record for this book is available from the British Library.

ISBN: 978-0-12-407192-6

Printed and bound in the United States of America

14 15 16 17 18 10 9 8 7 6 5 4 3 2 1



For information on all MK publications visit our website at *www.mkp.com*

Dedication

In praise of *Joe Celko's Complete Guide to NoSQL: What Every SQL Professional Needs to Know about Nonrelational Databases*

"For those of you who have problems that just don't fit the SQL mold, or who want to simply increase your knowledge of data management in general, you can do worse than Joe Celko's books in general, and NoSQL in particular."

—Jeff Garbus, Owner, Soaring Eagle Consulting

About the Author

Joe Celko served 10 years on the ANSI/ISO SQL Standards Committee and contributed to the SQL-89 and SQL-92 standards.

Mr. Celko is the author of a series of books on SQL and RDBMS for Elsevier/Morgan Kaufmann. He is an independent consultant based in Austin, TX. He has written over 1,200 columns in the computer trade and academic presses, mostly dealing with data and databases.

Introduction

> *"Nothing is more difficult than to introduce a new order, because the innovator has for enemies all those who have done well under the old conditions and lukewarm defenders in those who may do well under the new." —Niccolo Machiavelli*

I have done a series of books for the Elsevier/Morgan Kaufmann imprint over the last few decades. They have almost all been about SQL and RDBMS. This book is an overview of what is being called *Big Data*, *new SQL*, or *NoSQL* in the trade press; we geeks love buzzwords! The first columnist or blogger to invent a meme that catches on will have a place in Wikipedia and might even get a book deal out of it.

Since SQL is the de facto dominate database model on Earth, anything different has to be positioned as a challenger. But what buzzwords can we use? We have had petabytes of data in SQL for years, so "Big Data" does not seem right. SQL has been evolving with a new ANSI/ISO standard being issued every five or so years, rather than the "old SQL" suddenly changing into "new SQL" overnight. That last meme makes me think of New Coke® and does not inspire confidence and success.

Among the current crop of buzzwords, I like "NoSQL" the best because I read it as "N. O. SQL," a shorthand for "not only SQL" instead of "no SQL," as it is often read. This implies that the last 40-plus years of database technology have done no good. Not true! Too often SQL people, me especially, become the proverbial "kid with a hammer who thinks every problem is a nail" when we are doing IT. But it takes more than a hammer to build a house.

Some of the database tools we can use have been around for decades and even predate RDBMS. Some of the tools are new because technology made them possible. When you open your toolbox, consider all of the options and how they fit the task.

This survey book takes a quick look at the old technologies that you might not know or have forgotten. Then we get to the "new stuff" and why it exists.

I am not so interested in hardware or going into particular software in depth. For one thing, I do not have the space and you can get a book with a narrow focus for yourself and your projects. Think of this book as a department-store catalog where you can go to get ideas and learn a few new buzzwords.

Please send corrections and comments to jcelko212@earthlink.net and look for feedback on the companion website (http://elsevierdirect.com/v2/companion.jsp?ISBN=9780124071926).

The following is a quick breakdown of what you can expect to find in this book:

Chapter 1: NoSQL and Transaction Processing. A queue of jobs being read into a mainframe computer is still how the bulk of commercial data processing is done. Even transaction processing models finish with a batch job to load the databases with their new ETL tools. We need to understand both of these models and how they can be used with new technologies.

Chapter 2: Columnar Databases. Columnar databases use traditional structured data and often run some version of SQL; the difference is in how they store the data. The traditional row-oriented approach is replaced by putting data in columns that can be assembled back into the familiar rows of an RDBMS model. Since columns are drawn from one and only one data type and domain, they can be compressed and distributed over storage systems, such as RAID.

Chapter 3: Graph Databases. Graph databases are based on graph theory, a branch of discrete mathematics. They model *relationships among entities* rather than doing computations and aggregations of the values of the attributes and retrievals based on those values.

Chapter 4: MapReduce Model. The MapReduce model is the most popular of what is generally called NoSQL or Big Data in the IT trade press. It is intended for fast retrieval of large amounts of data from large file systems in parallel. These systems trade this speed and volume for less data integrity. Their basic operations are simple and do little optimization. But a lot of applications are willing to make that trade-off.

Chapter 5: Streaming Databases and Complex Events. The relational model and the prior traditional database systems assume that the tables are static during a query and that the result is also a static table. But streaming databases are built on a model of constantly flowing data—think of river or a pipe of data moving in time. The best-known examples of streaming

data are stock and commodity trading done by software in subsecond trades. The system has to take actions based on events in this stream.

Chapter 6: Key–Value Stores. A key–value store is a collection of pairs, (< key >, < value >), that generalize a simple array. The keys are unique within the collection and can be of any data type that can be tested for equality. This is a form of the MapReduce family, but performance depends on how carefully the keys are designed. Hashing becomes an important technique.

Schema versus No Schema. SQL and all prior databases use a schema that defines their structure, constraints, defaults, and so forth. But there is overhead in using and maintaining schema. Having no schema puts all of the data integrity (if any!) in the application. Likewise, the presentation layer has no way to know what will come back to it. These systems are optimized for retrieval, and the safety and query power of SQL systems is replaced by better scalability and performance for retrieval.

Chapter 7: Textbases. The most important business data is not in databases or files; it is in text. It is in contracts, warranties, correspondence, manuals, and reference material. Text by its nature is fuzzy and bulky; traditional data is encoded to be precise and compact. Originally, textbases could only find documents, but with improved algorithms, we are getting to the point of reading and understanding the text.

Chapter 8: Geographical Data. Geographic information systems (GISs) are databases for geographical, geospatial, or spatiotemporal (space–time) data. This is more than cartography. We are not just trying to locate something on a map; we are trying to find quantities, densities, and contents of things within an area, changes over time, and so forth.

Chapter 9: Big Data and Cloud Computing. The term *Big Data* was invented by Forrester Research in a whitepaper along with the the four V buzzwords: volume, velocity, variety, and variability. It has come to apply to an environment that uses a mix of the database models we have discussed and tries to coordinate them.

Chapter 10: Biometrics, Fingerprints, and Specialized Databases. Biometrics fall outside commercial use. They identify a person *as a biological entity* rather than a commercial entity. We are now in the worlds of medicine and law enforcement. Eventually, however, biometrics may move into the commercial world as security becomes an issue and we are willing to trade privacy for security.

Chapter 11: Analytic Databases. The traditional SQL database is used for online transaction processing. Its purpose is to provide support for daily business applications. The online analytical processing databases are built on the OLTP data, but the purpose of this model is to run queries that deal with aggregations of data rather than individual transactions. It is analytical, not transactional.

Chapter 12: Multivalued or NFNF Databases. RDBMS is based on first normal form, which assumes that data is kept in scalar values in columns that are kept in rows and those records have the same structure. The multivalued model allows the use to nest tables inside columns. They have a niche market that is not well known to SQL programmers. There is an algebra for this data model that is just as sound as the relational model.

Chapter 13: Hierarchical and Network Database Systems. IMS and IDMS are the most important prerelational technologies that are still in wide use today. In fact, there is a good chance that IMS databases still hold more commercial data than SQL databases. These products still do the “heavy lifting” in banking, insurance, and large commercial applications on mainframe computers, and they use COBOL. They are great for situations that do not change much and need to move around a lot of data. Because so much data still lives in them, you have to at least know the basics of hierarchical and network database systems to get to the data to put it in a NoSQL tool.

CHAPTER 1

NoSQL and Transaction Processing

Abstract

This chapter discusses traditional batch and transaction processing. A queue of jobs being read into a mainframe computer is how the bulk of commercial data processing is still done. Even transaction processing models finish with a batch job to load the databases with their new ETL tools. We need to understand both of these models and how they can be used with new technologies, and this chapter discusses these details.

Keywords

NoSQL; transaction processing

Introduction

This chapter discusses traditional batch and transaction processing. A queue of jobs being read into a mainframe computer is still how the bulk of commercial data processing is done. Even transaction processing models finish with a batch job to load the databases with their new ETL (extract, transform, load) tools. We need to understand both of these models and how they can be used with new technologies.

In the beginning, computer systems did monoprocessing, by which I mean they did one job from start to finish in a sequence. Later, more than one job could share the machine and we had multiprocessing. Each job was still independent of the others and waited in a queue for its turn at the hardware.

This evolved into a transaction model and became the client–server architecture we use in SQL databases. The goal of a transactional system is to assure particular kinds of data integrity are in place at the end of a transaction. NoSQL does not work that way.

1.1 Databases Transaction Processing in the Batch Processing World

Let’s start with a historical look at data and how it changed. Before there was Big Data there was “Big Iron”—that is, the mainframe computers, which used batch processing. Each program ran with its own unshared data and unshared processor; there was no conflict with other users or resources. A magnetic tape or deck of punch cards could be read by only one job at a time.

Scheduling batch jobs was an external activity. You submitted a job, it went into a queue, and a scheduler decided when it would be run. The system told an operator (yes, this is a separate job!) which tapes to hang on, what paper forms to load into a printer, and all the physical details. When a job was finished, the scheduler had to release the resources so following jobs could use them.

The scheduler had to assure that every job in the queue could finish. A job might not finish if other jobs were constantly assigned higher priority in the queue. This is called a *live-lock* problem. Think of the runt of a litter of pigs always being pushed away from its mother by the other piglets. One solution is to decrease the priority number of a job when it has been waiting for *n* seconds in the queue until it eventually gets to the first position.

For example, if two jobs, J1 and J2, both want to use resources A and B, we can get a dead-lock situation. Job J1 grabs resource A and waits for resource B; job J2 grabs resource B and waits for resource A. They sit and wait forever, unless one or both of the jobs releases its resource or we can find another copy of one of the resources.

This is still how most commercial data processing is done, but the tape drives have been swapped for disk drives.

1.2 Transaction Processing in the Disk Processing World

The world changed when disk drives were invented. At first, they were treated like fast tape drives and were mounted and dismounted and assigned to a single job. But the point of a database is that it is a common resource with multiple jobs (sessions) running at the same time.

There is no queue in this model. A user logs on in a session, which is connected to the entire database. Tables are not files, and the user does not connect to a particular table. The Data Control Language (DCL) inside the SQL engine decides what tables are accessible to which users.

If the batch systems were like a doorman at a fancy nightclub, deciding

who gets inside, then a database system is like a waiter handling a room full of tables (sorry, had to do that pun) that are concurrently doing their own things.

In this world, the amount of data available to a user session is huge compared to a magnetic tape being read record by record. There can be many sessions running at the same time. Handling that traffic is a major conceptual and physical change.

1.3 ACID

The late Jim Gray really invented modern transaction processing in the 1970s and put it in the classic paper "The Transaction Concept: Virtues and Limitations" in June 1981. This is where the ACID (atomicity, consistency, isolation, and durability) acronym started. Gray's paper discussed atomicity, consistency, and durability; isolation was added later. Bruce Lindsay and colleagues wrote the paper "Notes on Distributed Databases" in 1979 that built upon Gray's work, and laid down the fundamentals for achieving consistency and the primary standards for database replication. In 1983, Andreas Reuter and Theo Härder published the paper "Principles of Transaction-Oriented Database Recovery" and coined the term *ACID*.

The terms in ACID mean:

- *Atomicity.* Either the task (or all tasks) within a transaction are performed or none of them are. This is the all-or-none principle. If one element of a transaction fails, the entire transaction fails. SQL carries this principle internally. An `INSERT` statement inserts an entire set of rows into a table; a `DELETE` statement deletes an entire set of rows from a table; an `UPDATE` statement deletes and inserts entire sets.
- *Consistency.* The transaction must meet all protocols or rules defined by the system at all times. The transaction does not violate those protocols and the database must remain in a consistent state at the beginning and end of a transaction. In SQL this means that all constraints are `TRUE` at the end of a transaction. This can be because the new state of the system is valid, or because the system was rolled back to its initial consistent state.
- *Isolation.* No transaction has access to any other transaction that is in an intermediate or unfinished state. Thus, each transaction is independent unto itself. This is required for both performance and consistency of transactions within a database. This is not true in SQL; we have a concept of levels of isolation. A session can see uncommitted data at certain levels

of isolation. This uncommitted data can be rolled back by its session, so in one sense, it never existed.

- *Durability.* Once the transaction is complete, it will persist as complete and cannot be undone; it will survive system failure, power loss, and other types of system breakdowns. This is a hardware problem and we have done a good job of this. We just do not let data sit in volatile storage before we persist it.

This has been done with various locking schemes in most SQL databases. A lock says how other sessions can use a resource, such as reading only committed rows, or allowing them to read uncommitted rows, etc. This is called a *pessimistic concurrency* model. The underlying assumption is that you have to protect yourself from other people and that conflict is the normal situation.

The other popular concurrency model is called *optimistic concurrency*. If you have worked with microfilm, you know this model. Everyone gets a copy of the data to do with it as they wish in their session. In microfilm systems, the records manager would make copies of a document from the film and hand them out. Each employee would mark up his or her copy and turn it into central records.

The assumptions in this model are:

- Queries are much more common than database changes. Design for them.
- Conflicts are rare when there are database changes. Treat them as exceptions.
- When you do have conflicts, the sessions involved can be rolled back or you can set up rules for resolutions. Wait for things to get back to normal, and do not panic.

In case of microfilm systems, most of the requests were for information and the data was never changed. The requests that did make changes were usually separated in time so they did not conflict. When one or more employees made the same change, there was no conflict and the change was made. When two employees had a conflict, the records manager rejected both changes. Then he or she waited for another change that had no conflicts either by applying a rule or by a later change.

Optimistic concurrency depends on timestamping each row and keeping generational copies. The user can query the database at a point in time when he or she knows it is in an ACID state. In terms of the microfilm analogy, this

is how central records look while waiting for the employees to return their marked-up copies. But this also means that we start with the database at time = t_0, and can see it at time = t_0, t_1, t_2, …, t_n as we wish, based on the timestamps. Insertions, deletes, and updates do not interfere with queries as locking can. Optimistic concurrency is useful in situations where there is a constant inflow of data that has to be queried, such as stock and commodity trading.

The details of optimistic concurrency will be discussed in Section 5.1.1 on streaming databases. This method is best suited for databases that have to deal with constantly changing data, but have to maintain data integrity and present a consistent view of the data at a point in time.

Notice what has not changed: central control of data!

1.4 Pessimistic Concurrency in Detail

Pessimistic concurrency control assumes that conflict is the expected condition and we have to guard against it. The most popular models in a relational database management system (RDBMS) have been based on locking. A lock is a device that gives one user session access to a resource while keeping or restricting other sessions from that resource. Each session can get a lock on resources, make changes and then `COMMIT` or `ROLLBACK` the work in the database. A `COMMIT` statement will persist the changes, and a `ROLLBACK` statement will restore the database to the state it was in before the session. The system can also do a `ROLLBACK` if there is a problem with the changes. At this point, the locks are released and other sessions can get to the tables or other resources.

There are variants of locking, but the basic SQL model has the following ways that one transaction can affect another:

- *P0 (dirty write).* Transaction T1 modifies a data item. Another transaction, T2, then further modifies that data item before T1 performs a `COMMIT` or `ROLLBACK`. If T1 or T2 then performs a `ROLLBACK`, it is unclear what the correct data value should be. One reason why dirty writes are bad is that they can violate database consistency. Assume there is a constraint between x and y (e.g., $x = y$), and T1 and T2 each maintain the consistency of the constraint if run alone. However, the constraint can easily be violated if the two transactions write x and y in different orders, which can only happen if there are dirty writes.

- *P1 (dirty read).* Transaction T1 modifies a row. Transaction T2 then reads that row before T1 performs a `COMMIT`. If T1 then performs a `ROLLBACK`, T2 will have read a row that was never committed and that may thus be considered to have never existed.
- *P2 (nonrepeatable read).* Transaction T1 reads a row. Transaction T2 then modifies or deletes that row and performs a `COMMIT`. If T1 then attempts to reread the row, it may receive the modified value or discover that the row has been deleted.
- *P3 (phantom).* Transaction T1 reads the set of rows *N* that satisfy some search condition. Transaction T2 then executes statements that generate one or more rows that satisfy the search condition used by transaction T1. If transaction T1 then repeats the initial read with the same search condition, it obtains a different collection of rows.
- *P4 (lost update).* The lost update anomaly occurs when transaction T1 reads a data item, T2 updates the data item (possibly based on a previous read), and then T1 (based on its earlier read value) updates the data item and performs a `COMMIT`.

These phenomena are not always bad things. If the database is being used only for queries, without any changes being made during the workday, then none of these problems will occur. The database system will run much faster if you do not have to try to protect yourself from these problems. They are also acceptable when changes are being made under certain circumstances.

Imagine that I have a table of all the cars in the world. I want to execute a query to find the average age of drivers of red sport cars. This query will take some time to run, and during that time, cars will be crashed, bought, and sold; new cars will be built; and so forth. But I accept a situation with the three phenomena (P1–P3) because the average age will not change that much from the time I start the query to the time it finishes. Changes after the second decimal place really do not matter.

You can prevent any of these phenomena by setting the transaction isolation levels. This is how the system will use locks. The original ANSI model included only P1, P2, and P3. The other definitions first appeared in Microsoft Research Technical Report MSR-TR-95-51: “A Critique of ANSI SQL Isolation Levels” by Hal Berenson and colleagues (1995).

1.4.1 Isolation Levels

In standard SQL, the user gets to set the isolation level of the transactions in his or her session. The isolation level avoids some of the phenomena we just talked about and gives other information to the database. The following is the syntax for the SET TRANSACTION statement:

```
SET TRANSACTION < transaction mode list >

< transaction mode > ::= < isolation level> | < transaction
  access mode> | < diagnostics size >

< diagnostics size > ::= DIAGNOSTICS SIZE < number of
  conditions >

< transaction access mode > ::= READ ONLY | READ WRITE

< isolation level > ::= ISOLATION LEVEL < level of isolation >

< level of isolation > ::=

  READ UNCOMMITTED

    | READ COMMITTED

    | REPEATABLE READ–

    | SERIALIZABLE
```

The optional < diagnostics size > clause tells the database to set up a list for error messages of a given size. This is a standard SQL feature, so you might not have it in your particular product. The reason is that a single statement can have several errors in it and the engine is supposed to find them all and report them in the diagnostics area via a GET DIAGNOSTICS statement in the host program.

The < transaction access mode > clause explains itself. The READ ONLY option means that this is a query and lets the SQL engine know that it can relax a bit. The READ WRITE option lets the SQL engine know that rows might be changed, and that it has to watch out for the three phenomena.

The important clause, which is implemented in most current SQL products, is < isolation level >. The isolation level of a transaction defines the degree to which the operations of one transaction are affected by concurrent transactions. The isolation level of a transaction is SERIALIZABLE by default, but the user can explicitly set it in the SET TRANSACTION statement.

The isolation levels each guarantee that each transaction will be executed completely or not at all, and that no updates will be lost. The SQL engine, when it detects the inability to guarantee the serializability of two or more concurrent transactions or when it detects unrecoverable errors, may initiate a

ROLLBACK statement on its own.

Let's take a look at Table 1.1, which shows the isolation levels and the three phenomena. A Yes means that the phenomena are possible under that isolation level.

Table 1.1

Isolation Levels and the Three Phenomena

Isolation Level	P1	P2	P3
SERIALIZABLE	No	No	No
REPEATABLE READ	No	No	Yes
READ COMMITTED	No	Yes	Yes
READ UNCOMMITTED	Yes	Yes	Yes

In the table:

- The SERIALIZABLE isolation level is guaranteed to produce the same results as the concurrent transactions would have had if they had been done in some serial order. A serial execution is one in which each transaction executes to completion before the next transaction begins. The users act as if they are standing in a line waiting to get complete access to the database.
- A REPEATABLE READ isolation level is guaranteed to maintain the same image of the database to the user during his or her session.
- A READ COMMITTED isolation level will let transactions in this session see rows that other transactions commit while this session is running.
- A READ UNCOMMITTED isolation level will let transactions in this session see rows that other transactions create without necessarily committing while this session is running.

Regardless of the isolation level of the transaction, phenomena P1, P2, and P3 shall not occur during the implied reading of schema definitions performed on behalf of executing a statement, the checking of integrity constraints, and the execution of referential actions associated with referential constraints. We do not want the schema itself changing on users.

1.4.2 Proprietary Isolation Levels

We have discussed the ANSI/ISO model, but vendors often implement proprietary isolation levels. You will need to know how those work to use your product. ANSI/ISO sets its levels at the session level for the entire

schema. Proprietary models might allow the programmer to assign locks at the table level with additional syntax. Microsoft has a list of hints that use the syntax:

```
SELECT.. FROM < base table > WITH (< hint list >)
```

The model can apply row or table level locks. If they are applied at the table, you can get ANSI/ISO conformance. For example, WITH (HOLDLOCK) is equivalent to SERIALIZABLE, but it applies only to the table or view for which it is specified and only for the duration of the transaction defined by the statement that it is used in.

The easiest way to explain the various schemes is with the concept of readers and writers. The names explain themselves.

In Oracle, writers block writers, The data will remain locked until you either COMMIT, ROLLBACK or stop the session without saving. When two users try to edit the same data at the same time, the data locks when the first user completes an operation. The lock continues to be held, even as this user is working on other data.

Readers do not block writers: Users reading the database do not prevent other users from modifying the same data at any isolation level.

DB2 and Informix are little different. Writers block writers, like Oracle. But in DB2 and Informix, writers prevent other users from reading the same data at any isolation level above UNCOMMITTED READ. At these higher isolation levels, locking data until edits are saved or rolled back can cause concurrency problems; while you’re working on an edit session, nobody else can read the data you have locked. editing.

Readers block writers: In DB2 and Informix, readers can prevent other users from modifying the same data at any isolation level above UNCOMMITTED READ. Readers can only truly block writers in an application that opens a cursor in the DBMS, fetches one row at a time, and iterates through the result set as it processes the data. In this case, DB2 and Informix start acquiring and holding locks as the result set is processed.

In PostgreSQL, a row cannot be updated until the first transaction that made a change to the row is either committed to the database or rolled back. When two users try to edit the same data at the same time, the first user blocks other updates on that row. Other users cannot edit that row until this user either saves, thereby committing the changes to the database, or stops the edit session without saving, which rolls back all the edits performed in that edit session. If you use PostgreSQL’s multiversion concurrency control

(MVCC), which is the default and recommended behavior for the database, user transactions that write to the database do not block readers from querying the database. This is true whether you use the default isolation level of READ COMMITTED in the database or set the isolation level to SERIALIZABLE. Readers do not block writers: No matter which isolation level you set in the database, readers do not lock data.

1.5 CAP Theorem

In 2000, Eric Brewer presented his keynote speech at the ACM Symposium on the Principles of Distributed Computing and introduced the CAP or Brewer's theorem. It was later revised and altered through the work of Seth Gilbert and Nancy Lynch of MIT in 2002, plus many others since.

This theorem is for distributed computing systems while traditional concurrency models assume a central concurrency manager. The pessimistic model had a traffic cop and the optimistic model had a head waiter. CAP stands for consistency, availability, and partition tolerance:

- *Consistency* is the same idea as we had in ACID. Does the system reliably follow the established rules for its data content? Do all nodes within a cluster see all the data they are supposed to? Do not think that this is so elementary that no database would violate it. There are security databases that actively lie to certain users! For example, when you and I log on to the Daily Plant database, we are told that Clark Kent is a mild-mannered reporter for a great metropolitan newspaper. But if you are Lois Lane, you are told that Clark Kent is Superman, a strange visitor from another planet.
- *Availability* means that the service or system is available when requested. Does each request get a response outside of failure or success? Can you log on and attach your session to the database?
- *Partition tolerance* or robustness means that a given system continues to operate even with data loss or system failure. A single node failure should not cause the entire system to collapse. I am looking at my three-legged cat—she is partition tolerant. If she was a horse, we would have to shoot her.

Distributed systems can only guarantee two of the features, not all three. If you need availability and partition tolerance, you might have to let consistency slip and forget about ACID. Essentially, the system says "I will get you to a node, but I do not know how good the data you find there will

be" or "I can be available and the data I show will be good, but not complete." This is like the old joke about software projects: you have it on time, in budget, or correct—pick two.

Why would we want to lose the previous advantages? We would love to have them, but "Big Iron" has been beaten out by Big Data and it is spread all over the world. There is no central computer; every enterprise has to deal with hundreds, thousands, or tens of thousands of data sources and users on networks today.

We have always had Big Data in the sense of a volume that is pushing the limitations of the hardware. The old joke in the glory days of the mainframe was that all you needed to do was buy more disks to solve any problem. Today, the data volume uses terms that did not exist in the old days. The SI prefixes peta (1015) and exa (1018) were approved in 1975 at the 15th Conférence Générale des Poids et Mesures (CGPM).

1.6 BASE

The world is now full of huge distributed computing systems, such as Google's BigTable, Amazon's Dynamo, and Facebook's Cassandra. Here is where we get to BASE, a deliberately cute acronym that is short for:

- *Basically available.* This means the system guarantees the availability of the data as per the CAP theorem. But the response can be "failure," "unreliable" because the requested data is in an inconsistent or changing state. Have you ever used a magic eight ball?
- *Soft state.* The state of the system could change over time, so even during times without input there may be changes going on due to "eventual consistency," thus the system is always assumed to be soft as opposed to hard, where the data is certain. Part of the system can have hard data, such as a table of constants like geographical locations.
- *Eventual consistency.* The system will eventually become consistent once it stops receiving input. This gives us a window of inconsistency that is acceptably short. The term *acceptably short window* is a bit vague. A data warehouse doing noncritical computations can wait, but an online order-taking system has to respond in time to keep the customers happy (less than one minute). At the other extreme, real-time control systems must respond instantly. The domain name system (DNS) is the most commonly known system that uses eventual consistency. Updates to a domain name are passed around with protocols and time-controlled caches; eventually,

all clients will see the update. But it is far from instantaneous or centralized. This model requires a global timestamp so that each node knows which data item is the most recent version.

Like the ACID model, the eventual consistency model has variations:

- *Causal consistency.* If process A has sent an update to process B, then a subsequent access by process B will return the updated value, and a write is guaranteed to supersede the earlier write. Access by process C that has no causal relationship to process A is subject to the normal eventual consistency rules. This was also called a *buddy system* in early network systems. If a node could not get to the definitive data source, it would ask a buddy if it had gotten the new data and trust its data.
- *Read-your-writes consistency.* Process A, after it has updated a data item, always accesses the updated value and will never see an older value. This is a special case of the causal consistency model.
- *Session consistency.* This is a practical version of the previous model, where a process accesses the storage system in a session. As long as the session exists, the system guarantees read-your-writes consistency. If the session terminates because of a failure, a new session will be created and processing will resume with a guarantee that it will not overlap the prior sessions.
- *Monotonic read consistency.* A process returns only the most recent data values; it never returns any previous values.
- *Monotonic write consistency.* In this case the system guarantees to serialize the writes by the same process. Systems that do not guarantee this level of consistency are notoriously hard to program. Think of it as a local queue at a node in the network.

A number of these properties can be combined. For example, one can get monotonic reads combined with session-level consistency. From a practical point of view monotonic reads and read-your-writes properties are most desirable in an eventual consistency system, but not always required. These two properties make it simpler for developers to build applications, while allowing the storage system to relax consistency and provide high availability.

Eventual consistency has been part of the backup systems in RDBMS products and in synchronous and asynchronous replication techniques. In synchronous mode the replica update is part of the transaction. In asynchronous mode the updates are delayed by log shipping. If the database

crashes before the logs are shipped, the backup data can be out of date or inconsistent. Basically, the inconsistency window depends on the frequency of the log shipping.

1.7 Server-side Consistency

On the server side we will have the same data in several, not necessarily all, nodes. If all *n* nodes agree on a data value, then we are sure of it. Life is good.

But when we are in the process of establishing a consensus on an update, we need to know how many nodes have acknowledged the receipt of the update so far out of the nodes that are on the mailing list. We are looking for a quorum rule that accounts for node failures and incomplete replication. These rules will vary with the application. Large bank transfers will probably want complete agreement on all nodes. An abandoned website shopping-cart application can be satisfied if the customer returns to any node to continue shopping with some version of his or her cart, even one with some missing items. You just have to be sure that when the user hits the “checkout” key the other nodes know to delete their local copy of that cart.

What we do not want is sudden emergency restarts of nodes as a default action. This was how early file systems worked. Many decades ago, my wife worked for an insurance company that processed social security data. A single bad punch card would abort the entire batch and issue a useless error message.

We want a system designed for graceful degradation. The Sabre airline reservation system expects a small number of duplicate reservations. It does not matter if somebody has two conflicting or redundant reservations in the system. Since the passenger cannot be in two places at once or in one place twice, the problem will be resolved by a human being or the physical logic of the problem.

When one node is overloaded, you will tolerate the reduced performance and move some of the load to other nodes until the first system can be repaired. The best example of that is redundant array of independent disks (RAID) systems. When a disk fails, it is physically removed from the array and a new unit is plugged in to replace it. During the reconstruction of the failed disk, performance for accesses will take a hit. The data has to be copied from the alternative array blocks while the system keeps running its usual tasks.

1.8 Error Handling

There are two broad classes of error messages. We can have an anticipated problem, like an invalid password, which can have a standard response or process. We all have gotten an invalid password, and then been locked out if we fail to get it right in some number of tries.

The second class of error message tells us what happened, perhaps in painful detail. This invites some action on the part of the user or lets the user know why he or she is stuck.

But with NoSQL and the eventual consistency model, things might not be comfortable. Things stop or lock and you have no idea why, what to do, or how long it will take to resolve (if ever). As of 2012, Twitter has been trying to move from MySQL to Cassandra for more than a year. There are people (i.e., tweeters) who want instant feedback on the people they follow and any delay becomes an eternity. In August 2011, Foursquare reported an 11-hour downtime because of a failure of MongoDB.

1.9 Why SQL Does Not Work Here

To summarize why you might want to break away from SQL and the traditional RDBMS model:

- You do not have the data in one machine or even one network.
- Lots of it is not your data at all.
- It is so big you cannot put it in one place.
- It is uncoordinated in time as well as space.
- It is not always nice, structured data that SQL was meant to handle. We will spend the next few chapters on the new flavors of data and the special tools they need.

Concluding Thoughts

You need to know the basics of the old technology to understand the new technology.

References

1. Gray J. *The Transaction Concept: Virtues and Limitations*. 1981;

http://www.hpl.hp.com/techreports/tandem/TR-81.3.pdf; 1981; Cupertino CA.

2. Berenson H, et al. *Microsoft Research Technical Report MSR-TR-95-51: "A critique of ANSI SQL isolation levels"*. 1995; Redmond, WA.

CHAPTER 2

Columnar Databases

Abstract

This chapter discusses columnar databases, which use a nontraditional storage model. Columnar databases use traditional structured data and often run some version of SQL. The difference is in how they store the data. The traditional row-oriented approach is replaced by putting data in columns that can be assembled back into the familiar rows of an RDBMS model. Since columns are drawn from one and only one data type and domain, they can be compressed for faster access, and columns are easier to share among multiple users and distribute over modern redundant hardware storage systems, such as RAID.

Keywords

columnar data storage; hashing; MPP (massively parallel processing); multidimensional database (MDB); OLAP (online analytical processing); RAID (redundant array of independent disks); row-based optimizers; SMP (symmetric multiple processing); Sybase IQ; text compression

Introduction

Since the days of punch cards and magnetic tapes, files have been physically contiguous bytes that are accessed from start (open file) to finish (end-of-file flag = `TRUE`). Yes, the storage could be split up on a disk and the assorted data pages connected by pointer chains, but it is still the same model. Then the file is broken into records (more physically contiguous bytes), and records are broken into fields (still more physically contiguous bytes).

A file is processed in record by record (read/fetch next) or sequentially navigated in terms of a physical storage location (go to end of file, go back/forward *n* records, follow a pointer chain, etc.). There is no parallelism in this model. There is also an assumption of a physical ordering of the records within the file and an ordering of fields within the records. A lot of time and resources have been spent sorting records to make this access practical; you did not do random access on a magnetic tape and you could not do it with a deck of punch cards.

When we got to RDBMS and SQL, this file system model was still the dominant mindset. Even Dr. Codd fell prey to it. He first had to have a `PRIMARY KEY` in all tables, which corresponded to the sort order of a sequential file. Later, he realized that a key is a key and there is no need to make one of them special in RDBMS. However, SQL had already incorporated the old terminology and the early SQL engines were built on existing file systems, so it stuck.

The columnar model takes a fundamentally different approach. But it is one that works nicely with SQL and the relational model. In RDBMS, a table is an unordered set of rows that have exactly the same kind of rows. A row is an unordered set of columns all of the same kind, each of which holds scalar values drawn from a known domain. You access the columns by name, not by a physical position in the storage, but you have the `"SELECT*"` and other shorthand conventions to save typing.

The logical model is as follows: a table is a set of rows with one and only one structure; a row is a set of columns; a column is a scalar value drawn from one and only one domain. Storage usually follows this pattern with conventional file systems, using files for tables, records for rows, and fields for columns. But that has nothing to do with physical storage.

In the columnar model, we take a table and store each column in its own structure. Rows and tables are reassembled from these rows. Looking at the usual picture of a table, you can see why they are called vertical storage as opposed to horizontal storage models.

2.1 History

Column stores and inverted or transposed files are not that new. TAXIR was the first column-oriented database storage system that was built in 1969 for biology. Statistics Canada implemented the RAPID system in 1976 and used it for processing and retrieval of the Canadian Census of Population and Housing, as well as several other statistical applications. RAPID was shared with other statistical organizations throughout the world and used widely in the 1980s. It continued to be used by Statistics Canada until the 1990s.

For many years, Sybase IQ was the only commercially available columnar DBMS. However, when OLAP (online analytical processing) became popular, other products saw that they could take some of the techniques used for cubes and rollups and apply them to more general databases.

Given a column that has only one kind of simple data type in it, you can do

a lot of compression. To reconstruct the rows, you have to break the pure set model and number the rows, as you did in a sequential file. Each row is reconstituted by looking at this row number. Contiguous row numbers with the same data value can be modeled as ranges: {`start_position`, `end_position`, `data_value`}. This is the simplest format of compression, but it is very powerful. Much data is encoded in discrete values, so there are often long runs of identical values in a column.

Then we can compress the data value using specialized routines built for just that domain and the data type it uses. It would be silly to have a domain token the same size or larger than the data value. The payoff comes with wider columns. Building a lookup table with a short token for a longer value is easy. Now the model changes to {`start_position`, `end_position`, `domain_token`} references {`domain_token`, `data_value`}.

For example, the area code in U.S. phone numbers use the regular expression of [2–9][0–9][0–9], which means that we can have at most 800 of them. Instead of three ASCII characters, we can use a SMALLINT or a BCD for the token and get a lookup table that is tiny. This example is not a big savings, but it adds up in terabyte databases. A stronger example would be city and town names in the United States; there are slightly over 30,000 municipalities as of 2012. You can store 65,535 tokens in a 2-byte integer; very few towns have a name of only two letters, and many of them are duplicates (before you ask, “Springfield” is the most common town name, which is why it is used on television shows, most famously *The Simpsons*). Likewise, a 2-byte integer can sequence almost 180 years of dates.

It also gives us a single place to add certain integrity constraints. For example, area codes 666 and 777 are valid areas codes but were not assigned as of 2012. Likewise, 555 is dummy phone exchange that you will see in movies; before all-digit dialing, “KLondike 5” was how you guaranteed the phone number could not be called. Listen for it in old movies, cartoons, and modern American television shows. You can simply leave invalid values out of the lookup table!

Freeform text compression is well known. For a database, we want a lossless compression—that means that when you decompress it, you get back everything you put in it. In music and video, we can trade some loss for speed or space without being hurt. The Lempel–Ziv (LZ) compression methods are among the most popular algorithms for lossless storage. DEFLATE is a variation on LZ optimized for decompression speed and compression ratio, but compression can be slow. DEFLATE is used in PKZIP, gzip, and PNG.

LZW (Lempel–Ziv–Welch) is used in GIF images. Also noteworthy are the LZR (LZ–Renau) methods, which serve as the basis of the Zip method.

LZ methods use a table of substitutions for repeated strings. The table itself is often Huffman encoded (e.g., SHRI, LZX). This works well for human languages because grammatical affixes and structure words (prepositions, conjunctions, articles, particle markers, etc.) make up the bulk of text. But encoding schemes also tend to follow patterns. A bank account number starts with the code for the American Bankers Association bank number in the United States. A product serial number includes a product line. Hierarchical and vector encoding schemes invite this kind of compression.

A lot of work has been done in recent years with minimal perfect hashing algorithms (see the “A Quick Look at Hashing” sidebar if you do not the technique). When the column is a relatively static set of strings, this is ideal. Any set of values, even those without common substrings, are hashed to a short, fixed-length data token that can be computed quickly.

The naive reaction to this model is that it is complicated, big, and slow. That is only half right; it is complicated compared to simple sequential field reading. But it is not slow. And it is not big.

At one point the computer processing unit (CPU) was the bottleneck, but then SMP (symmetric multiple processing), clusters, and MPP (massively parallel processing) gave us hundreds or thousands of CPUs running faster than was ever possible before. Boyle’s law is a bit of IT folklore that computer processing doubles in speed and halves in cost every 18 months. At the same time, data is getting bigger. A decade ago, 20 gigabytes was considered unmanageable; now small Internet startups are managing terabytes of data.

MPP uses many separate CPUs running in parallel to execute a single program. MPP is similar to SMP, but in SMP all the CPUs share the same memory, whereas in MPP systems, each CPU has its own memory. The trade-off is that MPP systems are more difficult to program because the processors have to communicate and coordinate with each other. On the other hand, SMP systems choke when all the CPUs attempt to access the same memory at once. However, these CPUs sit idle more often than not. This is due to the inability of the pre-CPU layers of memory—L2 (especially) and L1 caches—to throughput data rapidly. We are still moving complete records in a row-based model of data.

As an analogy, think about your personal electronics. If you want a few

tracks from an album, it is cheaper to purchase only those tracks from iTunes. When you want most of the tracks, you will save money and time by purchasing the whole album, even though you only play some of the tracks.

A query is not the same kind of search-and-retrieval problem. The query either wants a column or it does not, and you know what that column is when you compile the query. The speed comes from the fact that assembling the rows with modern processor speeds and large primary storage is immensely faster than reading a single byte from a moving disk. In 2012, IBM was able to reduce the size of tables in DB2 to one-third or smaller of the original tables. This savings keeps paying off because the {`domain_token`, `data_value`} is all that is read and they can be put in faster storage, such as solid-state disk (SSD).

The major significant difference between columnar- and row-based storage is that all the columns of a table are not stored in data pages. This eliminates much of the metadata that is stored on a data page. This metadata varies from product to product, but the most common metadata is the displacement of the rows (records) from the front of each data page and perhaps some other metadata about the data page. Within that there is metadata for the displacement of the columns from the front of each row and if the column is null. This is how the SQL data management system knows where to place the read–write head on the disk and what to return.

In particular, when early SQL products put a `VARCHAR(n)` column into a row, they would be allocated in the order of the columns in the `CREATE TABLE` statement. But this meant they would allocate storage for the full *n* characters, even if the data is shorter. The kludge we used was to put all the `VARCHAR(n)` columns at the end of the row in the DDL manually. Today, DB2 and Oracle do this internally—rearrange the columns on output and hide it from the user.

The columnar model can compress data in place or store a pointer to a list of strings, such as `first_name` with a lookup table {1 = Aaron, 2 = Abe, 3 = Albert, …, 9999 = Zebadiah, …}. But now there is a trade-off; we can store `VARCHAR(n)` as fixed length to speed up searching and not be hurt because the lookup table is small. Each lookup value appears only once, so we still have a relatively small amount of storage.

Obviously, these metadata maps had to be updated as records are inserted or deleted on the data page. Deletion is easy; the rows that are removed can be flagged immediately and ignored when the column is read. Insertions can be added to the end of the column structure. While this works, it is nice to have clusters of identical values to keep the size small and make searching by

the data values easier. There are utility programs to recover storage—columnar versions of the garbage-collection routines from other memory management systems.

Columnar databases can have indexes, but most of them do not. In effect, columnar storage itself is an index. The displacements also has to be handled by the indexes.

A Quick Look at Hashing

Indexing and pointer chains involve a physical search to locate data. Given a search key, you can traverse from one node in a pointer chain to another until you either find a node with the search value or find that it does not exist.

Hashing is a disk-access technique based on mathematics. You take the search key and put it into a formula. The formula returns a location in an array or lookup table. That location contains a physical storage address and you can then go directly to it to find the data in *one* disk access.

For larger amounts of data, hashing is much faster than indexing. Tree-structured indexes become deeper as the amount of data increases. The traversals eventually have to get to a leaf node in this tree to find the data. This can mean more disk accesses with larger amounts of data.

It is possible that two or more different search keys will produce the same hash key. This is called a *collision* or (more poetically) a *hash clash*. The search key can then be rehashed with another function; the second function if often a member of the same family of hashing functions, with some of the constants changed. There is proof for certain hashing functions that a maximum of five rehashings will produce unique results.

A hashing function that has no collisions is called a *perfect* hashing function. If the hashing function has no empty slots in the array, then it is *minimal*. A minimal perfect hashing function requires that the set of search values is fixed, so it can work for keywords in a compiler and similar data sets.

The basic tools for most hashing algorithms are:

◆ *Digit selection.* Given a search key, we pull some of the digits

from the number, perhaps rearraging them. If the population is randomly distributed, this is a good technique. This is actually used by department stores at the order pickup windows. The first letter of the last names clusters around certain letters (“S” for Smith; “J” for Jones, Johnson, etc., but almost nobody for “X”, “Y,” and “Z”); however, customer phone numbers are uniformly random in the last two digits.

◆ *Division.* The mod (< key >, *m*) function with a prime number (*m*) can be very a good hash function (Lum et al., 1971). It will give you a result between (0, (*m* − 1)). The TOTAL and IMAGE/3000 databases came with a list of large primes to allocate hash tables.

◆ *Multiplication.* This technique squares a key and then pulls out the middle digits. A five-digit key will produce a ten-digit square and you can use the middle three digits with good results. For example, $54{,}321^2 = 2{,}950{,}771{,}041$ and the middle digits are 077. The hash has to come from the middle or you will get too much clustering.

◆ *Folding.* This technique pulls out continuous subsets of digits of size *n* and simply adds them. For example, given a five-digit key, you can add all five digits to get an answer in the range ($0 \le \text{hash} \le 45$). If you used pairs of digits, the range would be ($0 \le \text{hash} \le 207$). This is a weak technique, so exclusive-OR might be used instead of arithmetic and it is generally used with another technique.

Collision resolution techniques are also varied. The most common one is the use of buckets. A bucket is a hash table location that can hold more than one value. The two basic approaches are as follows:

◆ **Open address.** This method tries to find a bucket in the hash table that is still open. The simplest approach is to start at the collision location and do a linear search for an open space from that point. Other similar techniques use more complicated functions than increments. Common functions are quadratics and pseudorandom number generators.

◆ **External chaining.** You can add the new value to a linked list. This list might be in the main storage or have parts of it on

> disk. But with the right choice of the main storage table size, the overflow to disk can be less than 15% of the size of the hash table in the main storage. This method can be very effective.

2.2 How It Works

Since all values in a columnar store are of the same type and drawn from the same domain, calculating where the *n*th row is located is easy. All the columns are in the same order as they were in the original row, so to assemble the *i*th row, you go to the *i*th position in the relevant column stores and concatenate them. In the phone number example, go to `area_codes`, `phone_exchange`, and `phone_nbr` column stores and look for the *i*th records in each in parallel.

The area codes are relatively small, so they will come back first, then the exchanges, and finally the phone numbers. When I first saw this in the Sand (nee Nucleus) database, it was surprising. The test data was a set of over 5 million rows of Los Angeles, CA, public data and was set up to step through the data slowly for monitoring. The result set appeared on the screen of the test machine in columns, not rows. Columns did not materialize in the result set in left-to-right order, either!

2.3 Query Optimizations

Some columnar databases use row-based optimizers, which negates many of the benefits of columnar storage. They materialize “rows” (comprising of only the columns of the query, in effect doing selection and projection) early in the query execution and process them with a row-oriented optimizer. Column-based optimizers are able to divide the selection and projection functions into separate operations, which is a version of the MapReduce algorithms (these algorithms will be explained later).

The goal is to do as much with just the row numbers as possible before looking up the actual data values. If you can gather the columns in parallel, so much the better. Obviously, projection will come first since having the data in columns has already done this. But selection needs to be applied as soon as possible.

Notice that I have talked about columns being drawn from a domain. Most of the joins done in actual databases are equijoins, which means the columns

in different tables are drawn from the same domain and matched on equal values. In particular, a `PRIMARY KEY` and its referencing `FOREIGN KEY` have to be in the same domain. The `PRIMARY KEY` column will contain unique values for its table, and the `FOREIGN KEY`s will probably be one to many.

We can add a table name to the columnar descriptor, making it into domain descriptors: {`table_name`, `start_position`, `end_position`, `data_value`}. This vector can be fairly compact; a schema will seldom have the 2-plus million tables that can be modeled with a simple integers. This structure makes certain joins into scans over a single domain structure. An index can locate the start of each table within the domain descriptor and access the tables involved in parallel.

2.4 Multiple Users and Hardware

One of the advantages of a columnar model is that if two or more users want to use a different subset of columns, they do not have to lock out each other. This design is made easier because of a disk storage method known as RAID (redundant array of independent disks, originally redundant array of inexpensive disks), which combines multiple disk drives into a logical unit. Data is stored in several patterns called *levels* that have different amounts of redundancy. The idea of the redundancy is that when one drive fails, the other drives can take over. When a replacement disk drive in put in the array, the data is replicated from the other disks in the array and the system is restored. The following are the various levels of RAID:

- RAID 0 (block-level striping without parity or mirroring) has no (or zero) redundancy. It provides improved performance and additional storage but no fault tolerance. It is a starting point for discussion.
- In RAID 1 (mirroring without parity or striping) data is written identically to two drives, thereby producing a mirrored set; the read request is serviced by either of the two drives containing the requested data, whichever one involves the least seek time plus rotational latency. This is also the pattern for Tandem's nonstop computing model. Stopping the machine required a special command—"Ambush"—that has to catch both data flows at the same critical point, so they would not automatically restart.
- In RAID 10 (mirroring and striping) data is written in stripes across primary disks that have been mirrored to the secondary disks. A typical RAID 10 configuration consists of four drives: two for striping and two

for mirroring. A RAID 10 configuration takes the best concepts of RAID 0 and RAID 1 and combines them.

- In RAID 2 (bit-level striping with dedicated Hamming-code parity) all disk spindle rotation is synchronized, and data is striped such that each sequential bit is on a different drive. Hamming-code parity is calculated across corresponding bits and stored on at least one parity drive. This theoretical RAID level is not used in practice.
- In RAID 3 (byte-level striping with dedicated parity) all disk spindle rotation is synchronized, and data is striped so each sequential byte is on a different drive. Parity is calculated across corresponding bytes and stored on a dedicated parity drive.
- RAID 4 (block-level striping with dedicated parity) is equivalent to RAID 5 except that all parity data is stored on a single drive. In this arrangement, files may be distributed between multiple drives. Each drive operates independently, allowing input/output (I/O) requests to be performed in parallel. Parallelism is a huge advantage for a database. Each session can access one copy of a heavily referenced table without locking or read head contention.
- RAID 5, RAID 6, and other patterns exist; many of them are marketing terms more than technology. The goal is to provide fault tolerance of drive failures, up to *n* disk drive failures or removals from the array. This makes larger RAID arrays practical, especially for high-availability systems. While this is nice for database people, we get more benefit from parallelism for queries.

2.5 Doing an ALTER Statement

`ALTER` statements change the structures in a schema. In the columnar model, `ADD COLUMN` and `DROP COLUMN` are pretty easy; a new columnar structure is created or an old one is removed from physical storage. In a row-oriented model, each row has to be compressed or expended with the alteration. The indexes will also have to be restructured.

Changing the data type is also harder in a traditional row-oriented database because of the same space problem. In the real world most of the alterations are to increase the physical storage of a column. Numbers are made greater, strings are made longer; only dates seem immune to expansion of a data value since the ISO-8601 has a fixed range of 0001-01-01 to 9999-12-31 in the standard.

In the columnar model, the changes are much easier. Copy the positional data into a new columnar descriptor and cast the old data value to the new data value. When you have the new columnar structure loaded, drop the old and add the new. None of the queries will have to be changed unless they have a data type–specific predicate (e.g., if a date became an integer, then `foobar_date < = CURRENT_TIMESTAMP` is not going to parse).

2.6 Data Warehouses and Columnar Databases

Data warehouses can move some workloads, when few columns are involved, to columnar databases for improved performance. Multidimensional databases (MDBs), or cubes, are separate physical structures that support very fast access to precomputed aggregate data. When a query asks for most columns of the MDB, the MDB will perform quite well relatively speaking.

The physical storage of these MDBs is a denormalized dimensional model that eliminates joins by storing the computations. However, MDBs get large and grow faster than expected as columns are added. The data in a MDB can be compressed in much the same way that it is in a columnar database, so it is relative easy to extract a subset of columns from a cube.

The best workload for columnar databases is queries that access less than all columns of the tables they use. In this case, less is more. The smaller the percentage of the row's bytes needed, the better the performance difference with columnar databases.

Concluding Thoughts

A lot of important workloads are column selective, and therefore benefit tremendously from this model. Columnar databases perform well with larger amounts of data, large scans, and I/O-bound queries. While providing performance benefits, they also have unique abilities to compress their data.

Columnar databases have been around for a while and did very well in their niche. But they made a leap into the current market for two reasons. First, improved hardware, SSD in particular, made the differences between primary and secondary storage less clear. When there was a sharp difference between primary and secondary storage, compressing and decompressing data in and out of secondary storage was overhead. In SSD, there is no difference. The second factor is better algorithms. We have gotten really good at specialized compression on one hand, but we also have parallel algorithms that are designed for columnar data stores.

Reference

1. Lum VY, Yuen PST, Dodd M. Key to Address Transform Technique: A Fundamental Performance Study on Large Existing Formatted Files. *Communications of the ACM* 1971;228–239.

CHAPTER 3

Graph Databases

Abstract

Graph databases model relationships, not data. Just as SQL and RDBMS are based on logic and set theory, graph databases are based on graph theory. The classic example of a popular graph problem is called the “Kevin Bacon problem” in the literature. Given actor Kevin Bacon, a graph database links him to anyone else in the movie industry either by a direct link (had some role in a movie with Kevin Bacon) or by a chain of links (had some role in a movie with someone who had some role in a movie with Kevin Bacon). However, other graph problems look for the shortest paths among a set of paths in a transportation network, the smallest set of nodes that cover a graph, and so forth. There is no ANSI/ISO standard language for graphs, but the most popular ones are Gremlin, Neo4j, and SPARQL. They tend to model their syntax after SQL and/or mathematics.

Keywords

graph theory; Gremlin; Kevin Bacon problem; Neo4j; SPARQL

Introduction

This chapter discusses graph databases, which are used to model relationships rather than traditional structured data. Graph databases have nothing to do with presentation graphics. Just as FORTRAN is based on algebra, and relational databases are based on sets, graph databases are based on graph theory, a branch of discrete mathematics. Here is another way we have turned a mind tool into a class of programming tools!

Graph databases are not network databases. Those were the prerelational databases that linked records with pointer chains that could be traversed record by record. IBM’s Information Management System (IMS) is one such tool still in wide use; it is a hierarchical access model. Integrated Database Management System (IDMS), TOTAL, and other products use more complex pointer structures (e.g., single-link lists, double-linked lists, junctions, etc.) to allow more general graphs than just a tree. These pointer structures are “hardwired” so that they can be navigated; they are the structure in which the data sits.

In a graph database, we are not trying to do arithmetic or statistics. We

want to see relationships in the data. Curt Monash the database expert and blogger (http://www.dbms2.com/, http://www.monash.com) coined the term for this kind of analysis: *relationship analytics*.

Programmers have had algebra in high school, and they may have had some exposure to naive set theory in high school or college. You can program FORTRAN and other procedural languages with high school–level algebra and only a math major needs advanced set theory, which deals with infinite sets (computers do not have infinite storage no matter what your boss thinks). But you cannot really understand RDBMS and SQL without naive set theory.

But only math majors seem to take a whole semester of graph theory. This is really too bad; naive graph theory has simple concepts and lots of everyday examples that anyone can understand. Oh, did I mention that it is also full of sudden surprises where simple things become nonpolynomial (NP)-complete problems? Let's try to make up that gap in your education.

In computer science, we use the "big O" notation, $O(n)$, to express how much effort it takes to run an algorithm as the size of the input, (n). For example, if we have a simple process that handles one record at a time, the $O(n)$ is linear; add one more record and it takes one more unit of execution time. But some algorithms increase in more complex ways. For example, sorting a file can be done in $O(n \log_2(n))$ time. Other algorithms can have a complexity that is a polynomial usually with squares and cubes. Then we get to the NP algorithms. They usually involve having to try all possible combinations to find a solution, so they have a factorial in their complexity, $O(n!)$.

NP complexity shows up in many of the classic graph theory problems. Each new edge or node added to the graph can result in more and more combinations. We often find that we look for near-optimal solutions instead of practical reasons.

3.1 Graph Theory Basics

A graph has two things in it. There are edges (or arcs) and nodes (or vertices); the edges are drawn as lines that connect nodes, which are drawn as dots or circles. That is it! Two parts! Do not be fooled; binary numbers only have two parts and we build computers with them.

3.1.1 Nodes

Nodes are abstractions. They usually (not always) model what would be an entity in RDBMS. In fact, some of the power of graph theory is that a node can model a subgraph. A node may or may not have "something inside it" (electrical components) in the model; it can just sit there (bus stop) or simply be (transition state). A node is not an object. Objects have methods and local data inside them. In a complex graph query, we might be looking for an unknown or even nonexistent node. For example, a bus stop with a Bulgarian barbeque stand might not exist. But a bus stop with a barbeque in a Bulgarian neighborhood might exist, and we not do know it until we connect many factors together (e.g., riders getting off at the Bulgarian Culture Center bus stop, restaurants or Bulgarian churches within *n* blocks of the bus stop, etc.). Other examples of graphs you might have seen are:

- *Schematic maps:* the nodes are the bus stops, towns, and so forth.
- *Circuit diagrams:* the nodes are electrical components.
- *State transitions:* the nodes are the states (yes, this can be modeled in SQL).

3.1.2 Edges

Edges or arcs connect nodes. We draw them as lines that may or may not have an arrow head on them to show a direction of flow. In schematic maps, the edges are the roads. They can have a distance or time on them. In the circuit diagrams, the edges are the wires that have resistance, voltage, etc. Likewise, the abstract state transitions are connected by edges that model the legal transition paths.

In one way, edges are more fun than nodes in graph theory. In RDBMS models of data, we have an unspecified single relationship between tables in the form of a `REFERENCES` clause. In a graph database, we can have multiple edges of different kinds between nodes. These can be of various strengths that we know (e.g., "I am your father, Luke," if you are a *Star Wars* fan; "is a pen pal of") and ones that we establish from other sources (e.g., "subscribes to the *Wall Street Journal*"; "friend of a friend of a friend"; "son of a son of a sailor," if you are a Jimmy Buffett fan).

At the highest level of abstraction an edge can be directed or undirected. In terms of maps, these are one-way streets; for state transitions, this is prior state–current state pairs and so forth. We tend to like undirected graphs since the math is easier and there is often an inverse relationship of some sort (e.g., "father–son" in the case of Luke Skywalker and Darth Vader).

Colored edges are literally colored lines on a display of a graph database. One color is used to show the same kind of edge, the classic “friend of a friend of a friend,.” or a Bacon(*n*) relationship (I will explain this shortly) used in social networks when they send you a “you might also know …” message and ask you to send an invitation to that person to make a direct connection.

Weighted edges have a measurement that can accumulate. In the case of a map—distances—the accumulation rule is additive; in the case of the Bacon(*n*) function it diminishes over distance (you may ask, “Who? Oh, I forgot about him!”).

3.1.3 Graph Structures

The bad news is that since graph theory is fairly new by mathematical standards, which means less than 500 years old, there are still lots of open problems and different authors will use different terminology. Let me introduce some basic terms that people generally agree on:

- A *null graph* is a set of nodes without any edges. A *complete graph* has an edge between every pair of nodes. Both of these extremes are pretty rare in graph databases.
- A *walk* is a sequence of edges that connect a set of nodes without repeating an edge.
- A *connected graph* is a set of nodes in which any two nodes can be reached by a walk.
- A *path* is a walk that goes through each node only once. If you have *n* nodes, you will have ($n - 1$) edges in the path.
- A *cycle* or *circuit* is a path that returns to where it started. In RDBMS, we do not like circular references because referential actions and data retrieval can hang in an endless loop. A *Hamiltonian circuit* is one that contains all nodes in a graph.
- A *tree* is a connected graph that has no cycles. I have a book on how to model trees and hierarchies in SQL (Celko, 2012). Thanks to hierarchical databases, we tend to think of a directed tree in which subordination starts at a root node and flows down to leaf nodes. But in graph databases, trees are not as obvious as an organizational chart, and finding them is a complicated problem. In particular, we can start at a node as the root and look for the minimal spanning tree. This is a subset of edges that give us

the shortest path from the root we picked to each node in the graph.

Very often we are missing the edges we need to find an answer. For example, we might see that two people got traffic tickets in front of a particular restaurant, but this means nothing until we look at their credit card bills and see that these two people had a meal together.

3.2 RDBMS Versus Graph Database

As a generalization, graph databases worry about relationships, while RDBMSs worry about data. RDBMSs have difficulties with complex graph theoretical analysis. It's easy to manage a graph where every path has length one; that is just a three-column table (node, edge, node). By doing self-joins, you can construct paths of length two, and so forth, called a breadth-first search. If you need a mental picture, think of an expanding search radius. You quickly get into Cartesian explosions for longer paths, and can get lost in an endless loop if there are cycles. Furthermore, it is extremely hard to write SQL for graph analysis if the path lengths are long, variable, or not known in advance.

3.3 Six Degrees of Kevin Bacon Problem

The game "Six Degrees of Kevin Bacon" was invented in 1994 by three Albright College students: Craig Fass, Brian Turtle, and Mike Ginelli. They were watching television movies when the film *Footloose* was followed by *The Air Up There*, which lead to the speculation that everyone in the movie industry was connected in some way to Kevin Bacon. Kevin Bacon himself was assigned the Bacon number 0; anyone who had been in a film with him has a Bacon number of 1; anyone who worked with that second person has a Bacon number of 2; and so forth. The goal is to look for the shortest path. As of mid-2011, the highest finite Bacon number reported by the Oracle of Bacon is 8.

This became a fad that finally resulted in the website "Oracle of Bacon" (*http://oracleofbacon.org*), which allows you to do online searches between any two actors in the Internet Movie Database (*www.imdb.com*). For example, Jack Nicholson was in *A Few Good Men* with Kevin Bacon, and Michelle Pfeiffer was in *Wolf* with Jack Nicholson. I wrote a whitepaper for Cogito, Inc. of Draper, UT, in which I wrote SQL queries to the Kevin Bacon problem as a benchmark against their graph database. I want to talk about that in more detail.

3.3.1 Adjacency List Model for General Graphs

Following is a typical adjacency list model of a general graph with one kind of edge that is understood from context. Structure goes in one table and the nodes in a separate table, because they are separate kinds of things (i.e., entities and relationships). The SAG card number refers to the Screen Actors Guild membership identifier, but I am going to pretend that they are single letters in the following examples.

```
CREATE TABLE Actors
(sag_card CHAR(9) NOT NULL PRIMARY KEY
  actor_name VARCHAR(30) NOT NULL);
CREATE TABLE MovieCasts
(begin_sag_card CHAR(9) NOT NULL
  REFERENCES Nodes (sag_card)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
  end_sag_card CHAR(9) NOT NULL
  REFERENCES Nodes (sag_card)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
  PRIMARY KEY (begin_sag_card, end_sag_card),
  CHECK (begin_sag_card <> end_sag_card));
```

I am looking for a path from Kevin Bacon, who is ‘s‘ for “start” in the example data, to some other actor who has a length less than six. Actually, what I would really like is the shortest path within the set of paths between actors.

The advantage of SQL is that it is a declarative, set-oriented language. When you specify a rule for a path, you get *all* the paths in the set. That is a good thing—usually. However, it also means that you have to compute and reject or accept all possible candidate paths. This means the number of combinations you have to look at increases so fast that the time required to process them is beyond the computing capacity in the universe. It would be nice if there were some heuristics to remove dead-end searches, but there are not.

I made one decision that will be important later; I added self-traversal edges (i.e., an actor is always in a movie with himself) with zero length. I am

going to use letters instead of actor names. There are a mere five actors called {‘s’, ‘u’, ‘v’, ‘x’, ‘y’}:

```
INSERT INTO Movies – 15 edges
VALUES (’s’, ‘s’), (’s’, ‘u’), (’s’, ‘x’),
  (’u’, ‘u’), (’u’, ‘v’), (’u’, ‘x’), (’v’, ‘v’), (’v’, ‘y’), (’x’, ‘u’),
  (’x’, ‘v’), (’x’, ‘x’), (’x’, ‘y’), (’y’, ‘s’), (’y’, ‘v’), (’y’, ‘y’);
```

I am not happy about this approach, because I have to decide the maximum number of edges in the path before I start looking for an answer. But this will work, and I know that a path will have no more than the total number of nodes in the graph. Let’s create a query of the paths:

```
CREATE TABLE Movies
(in_node CHAR(1) NOT NULL,
  out_node CHAR(1) NOT NULL)
INSERT INTO Movies
VALUES (’s’, ‘s’), (’s’, ‘u’), (’s’, ‘x’),
  (’u’, ‘u’), (’u’, ‘v’), (’u’, ‘x’), (’v’, ‘v’),
  (’v’, ‘y’), (’x’, ‘u’), (’x’, ‘v’), (’x’, ‘x’),
  (’x’, ‘y’), (’y’, ‘s’), (’y’, ‘v’, (’y’, ‘y’);
CREATE TABLE Paths
(step1 CHAR(2) NOT NULL,
  step2 CHAR(2) NOT NULL,
  step3 CHAR(2) NOT NULL,
  step4 CHAR(2) NOT NULL,
  step5 CHAR(2) NOT NULL,
  path_length INTEGER NOT NULL,
PRIMARY KEY (step1, step2, step3, step4, step5));
```

Let’s go to the query and load the table with all the possible paths of length five or less:

```
DELETE FROM Paths;
INSERT INTO Paths
SELECT DISTINCT M1.out_node AS s1, – it is ‘s‘ in this example
  M2.out_node AS s2,
```

```
  M3.out_node AS s3,
  M4.out_node AS s4,
  M5.out_node AS s5,
  (CASE WHEN M1.out_node NOT IN (M2.out_node, M3.out_node,
    M4.out_node, M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M2.out_node NOT IN (M3.out_node, M4.out_node,
    M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M3.out_node NOT IN (M2.out_node, M4.out_node,
    M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M4.out_node NOT IN (M2.out_node, M3.out_node,
    M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M5.out_node NOT IN (
M2.out_node, M3.out_node, M4.out_node) THEN 1 ELSE 0 END)
  AS path_length
  FROM Movies AS M1, Movies AS M2, Movies AS M3, Movies AS M4,
    Movies AS M5
  WHERE M1.in_node = M2.out_node
  AND M2.in_node = M3.out_node
  AND M3.in_node = M4.out_node
  AND M4.in_node = M5.out_node
  AND 0 < (CASE WHEN M1.out_node NOT IN (M2.out_node,
    M3.out_node, M4.out_node, M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M2.out_node NOT IN (M1.out_node, M3.out_node,
    M4.out_node, M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M3.out_node NOT IN (M1.out_node, M2.out_node,
    M4.out_node, M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M4.out_node NOT IN (M1.out_node, M2.out_node,
    M3.out_node, M5.out_node) THEN 1 ELSE 0 END
  + CASE WHEN M5.out_node NOT IN (M1.out_node, M2.out_node,
    M3.out_node, M4.out_node) THEN 1 ELSE 0 END);
SELECT * FROM Paths ORDER BY step1, step5, path_length;
```

The Paths. `step1 column` is where the path begins. The other columns of Paths are the second step, third step, fourth step, and so forth. The last step column is the end of the journey. The `SELECT DISTINCT` is a safety thing and the “greater than zero” is to clean out the zero-length start-to-start paths. This is a complex query, even by my standards.

The path length calculation is a bit harder. This sum of CASE expressions looks at each node in the path. If it is unique within the row, it is assigned a value of 1; if it is not unique within the row, it is assigned a value of 0.

There are 306 rows in the path table. But how many of these rows are actually the same path? SQL has to have a fixed number of columns in a table, but paths can be of different lengths. That is to say that (*s*, *y*, *y*, *y*, *y*) = (*s*, *s*, *y*, *y*, *y*) = (*s*, *s*, *s*, *y*, *y*) = (*s*, *s*, *s*, *s*, *y*). A path is not supposed to have cycles in it, so you need to filter the answers. The only places for this are in the WHERE clause or outside of SQL in a procedural language.

Frankly, I found it was easier to do the filtering in a procedural language instead of SQL. Load each row into a linked list structure and use recursive code to find cycles. If you do it in SQL, you need a predicate for all possible cycles of size 1, 2, and so forth, up to the number of nodes in the graph.

Internally, graph databases will also use a simple (node, edge, node) storage model, but they will additionally add pointers to link nearby nodes or subgraphs. I did a benchmark against a "Kevin Bacon" database. One test was to find the degrees with Kevin Bacon as "the center of the universe," and then a second test was to find a relationship between any two actors. I used 2,674,732 rows of data. Ignoring the time to set up the data, the query times for the simple Bacon numbers are given in Table 3.1. The timings are raw clock times starting with an empty cache running on the same hardware. The SQL was Microsoft SQL Server, but similar results were later obtained with DB2 and Oracle.

Table 3.1

Query Times for Bacon Numbers

Bacon Number	SQL	Cogito
1	00:00:24	0.172 ms
2	00:02:06	00:00:13
3	00:12:52	00:00:01
4	00:14:03	00:00:13
5	00:14:55	00:00:16
6	00:14:47	00:00:43

The figures became much worse for SQL as I generalized the search (e.g., change the focus actor, use only actress links, use one common movie, and add directors). For example, changing the focus actor could be up to 9,000 times slower, most by several hours versus less than one minute.

3.3.2 Covering Paths Model for General Graphs

What if we attempt to store all the paths in a directed graph in a single table in an RDBMS? The table for this would look like the following:

```
CREATE TABLE Paths
(path_nbr INTEGER NOT NULL,
  step_nbr INTEGER NOT NULL
  CHECK (path_nbr > = 0),
  node_id CHAR(1) NOT NULL,
  PRIMARY KEY (path_nbr, step_nbr));
```

Each path is assigned an ID number and the steps are numbered from 0 (the start of the path) to *k*, the final step. Using the simple six-node graph, the one-edge paths are:

```
1 0 A
1 1 B
2 0 B
2 1 F
3 0 C
3 1 D
4 0 B
4 1 D
5 0 D
5 1 E
```

Now we can add the two-edge paths:

```
6 0 A
6 1 B
6 2 F
7 0 A
7 1 B
7 2 D
8 0 A
8 1 C
8 2 D
9 0 B
```

```
9 1 D
9 2 E
```

And finally the three-edge paths:

```
10 0 A
10 1 B
10 2 D
10 3 E
11 0 A
11 1 B
11 2 D
11 3 E
```

These rows can be generated from the single-edge paths using a common table expression (CTE) or with a loop in a procedural language, such as SQL/PSM. Obviously, there are fewer longer paths, but as the number of edges increases, so does the number of paths. By the time you get to a realistic-size graph, the number of rows is huge. However, it is easy to find a path between two nodes, as follows:

```
SELECT DISTINCT :in_start_node, :in_end_node,
  (P2.step_nbr- P1.step_nbr) AS distance
  FROM Paths AS P1, Paths AS P2
  WHERE P1.path_nbr = P2.path_nbr
  AND P1.step_nbr < = P2.step_nbr
  AND P1.node_id = :in_start_node
  AND P2.node_id = :in_end_node;
```

Notice the use of `SELECT DISTINCT` because most paths will be a subpath of one or more longer paths. Without it, the search for all paths from `A` to `D` in this simple graph would return:

```
7 0 A
7 1 B
7 2 D
8 0 A
8 1 C
8 2 D
```

```
10 0 A
10 1 B
10 2 D
11 0 A
11 1 B
11 2 D
```

However, there are only two distinct paths, namely (`A`, `B`, `D`) and (`A`, `C`, `D`). In a realistic graph with lots of connections, there is a good chance that a large percentage of the table will be returned.

Can we do anything to avoid the size problems? Yes and no. In this graph, most of the paths are redundant and can be removed. Look for a set of subpaths that cover all of the paths in the original graph. This is easy enough to do by hand for this simple graph:

```
1 0 A
1 1 B
1 2 F
2 0 A
2 1 B
2 2 D
2 3 E
3 0 A
3 1 C
3 2 D
3 3 E
```

The problem of finding the longest path in a general graph is known to be NP-complete, and finding the longest path is the first step of finding a minimal covering path set. For those of you without a computer science degree, NP-complete problems are those that require drastically more resources (storage space and/or time) as the number of elements in the problem increases. There is usually an exhaustive search or combinatory explosion in these problems.

While search queries are easier in this model, dropping, changing, or adding a single edge can alter the entire structure, forcing us to rebuild the entire table. The combinatory explosion problem shows up again, so loading and validating the table takes too long for even a medium number of nodes. In

another example, MyFamily.com (owner of Ancestry.com) wanted to let visitors find relationships between famous people and themselves. This involves looking for paths 10 to 20 + edges long, on a graph with over 200 million nodes and 1 billion edges. Query rates are on the order of 20 per second, or 2 million per day.

3.3.3 Real-World Data Has Mixed Relationships

Now consider another kind of data. You are a cop on a crime scene investigator show. All you have is a collection of odd facts that do not fall into nice, neat relational tables. These facts tie various data elements together in various ways. You now have 60 minutes to find a network of associations to connect the bad guys to the crime in some as-of-yet-unknown manner.

Ideally, you would do a join between a table of “bad guys” and a table of “suspicious activities” on a known relationship. You have to know that such a join is possible before you can write the code. You have to insert the data into those tables as you collect it. You cannot whip up another relationship on-the-fly.

Let’s consider an actual example. The police collect surveillance data in the form of notes and police reports. There is no fixed structure in which to fit this data. For example, U-Haul reports that a truck has not been returned and they file a police report. That same week, a farm-supply company reports someone purchased a large amount of ammonium nitrate fertilizer. If the same person did both actions, and used his own name (or with a known alias) in both cases, then you could join them into a relationship based on the “bad guys” table. This would be fairly easy; you would have this kind of query in a view for simple weekly reports. This is basically a shortest-path problem and it means that you are trying to find the dumbest terrorist in the United States.

In the real world, conspirator A rents the truck and conspirator B buys the fertilizer. Or one guy rents a truck and cannot return it on time while another totally unrelated person buys fertilizer paying in cash rather than using an account that is known to belong to a farmer. Who knows? To find if you have a coincidence or a conspiracy, you need a relationship between the people involved. That relationship can be weak (both people live in New York state) or strong (they were cellmates in prison).

Figure 3.1 is a screenshot of this query and the subgraph that answers it. Look at the graph that was generated from the sample data when it was actually given a missing rental truck and a fertilizer purchase. The result is a

network that joins the truck to the fertilizer via two ex-cons, who shared jail time and a visit to a dam. Hey, that is a red flag for anyone! This kind of graph network is called a *causal diagram* in statistics and fuzzy logic. You will also see the same approach as a fishbone diagram (also known as cause-and-effect diagram and Ishikawa diagram after their inventor) when you are looking for patterns in data. Before now, this method has been a "scratch-paper" technique. This is fine when you are working on one very dramatic case in a 60-minute police show and have a scriptwriter.

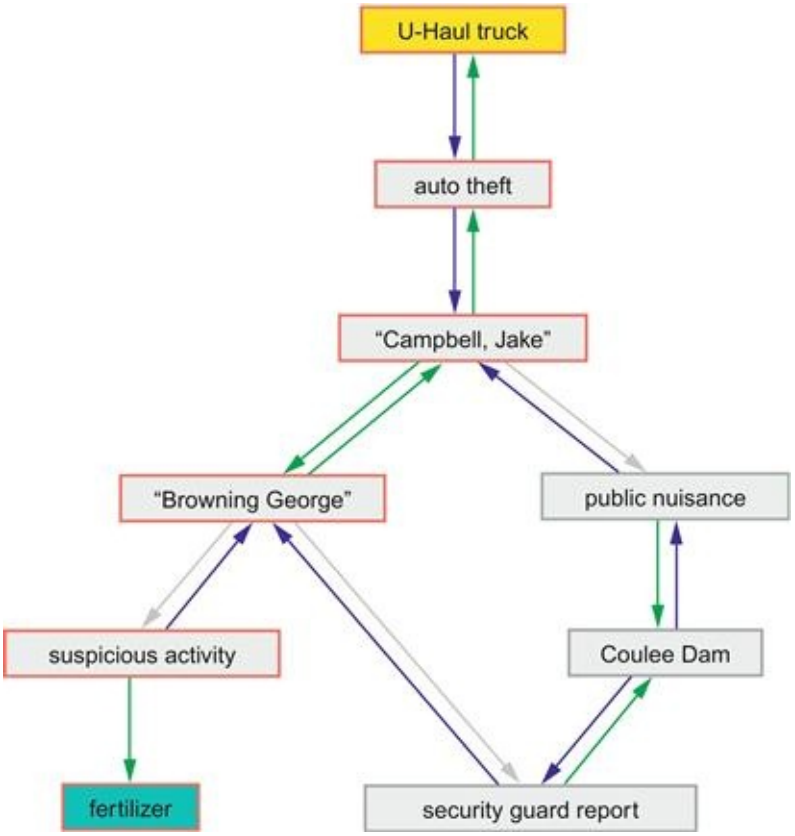


FIGURE 3.1 Possible Terrorist Attack Graph.

In the real world, a major police department has a few hundred cases a week. The super-genius Sherlock Holmes characters are few and far between. But even if you could find such geniuses, you simply do not have enough whiteboards to do this kind of analysis one case at a time in the real world. Intelligence must be computerized in the 21st century if it is going to work.

Most crime is committed by repeat offenders. Repeat offenders tend to follow patterns—some of which are pretty horrible, if you look at serial killers. What a police department wants to do is describe a case, then look through all the open cases to see if there are three or more cases that have the same pattern.

One major advantage is that data goes directly into the graph, while SQL

requires that each new fact has to be checked against the existing data. Then the SQL data has to be encoded on some scale to fit into a column with a particular data type.

3.4 Vertex Covering

Social network marketing depends on finding "the cool kids," the trendsetters who know everybody in a community. In some cases, it might be one person. The Pope is fairly well known among Catholics, for example, and his opinions carry weight.

Formally, a vertex cover of an undirecterd graph G is a set C of vertices such that each edge of G is incident to at least one vertex in C. The set C is said to cover the edges of G. Informally, you have a weird street map and want to put up security cameras at intersections (nodes) in such a way that no street (edge) is not under surveillance. We also talk about coloring the nodes to mark them as members of the set of desired vertices. Figure 3.2 shows two vertex coverings taken from a Wikipedia article on this topic.

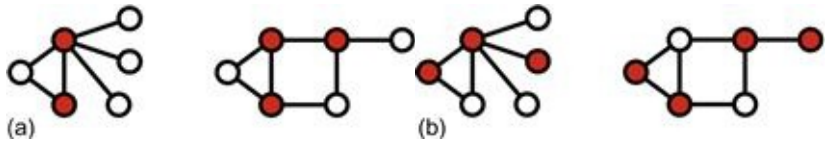


FIGURE 3.2 Vertex coverings from Wikipedia: (a) three-node solution, and (b) a four-node solution.

However, neither of these coverings is minimal. The three-node solution can be reduced to two nodes, and the four-node solution can be reduced to three nodes, as follows:

```
CREATE TABLE Grid

(v1 SMALLINT NOT NULL CHECK (v1 > 0),

  v2 SMALLINT NOT NULL CHECK (v2 > 0),

  PRIMARY KEY (v1, v2),

CHECK (v1 < v2),

  color SMALLINT DEFAULT 0 NOT NULL CHECK (color > = 0));

INSERT INTO Grid (v1, v2)

VALUES (1, 2), (1, 4), (2, 3), (2, 5), (2, 6);
```

{1, 2, 6} and {2, 4} are vertex covers. The second is the minimal cover. Can you prove it? In this example, you can use brute force and try all possible coverings. Finding a vertex covering is known to be an NP-complete problem,

so brute force is the only sure way. In practice, this is not a good solution because the combinatorial explosion catches up with you sooner than you think.

One approach is to estimate the size of the cover, then pick random sets of that size. You then keep the best candidates, look for common subgraphs, and modify the candidates by adding or removing nodes. Obviously, when you have covered 100% of the edges, you have a winner; it might not be optimal, but it works.

Another consideration is that the problem might start with a known number of nodes. For example, you want to give *n* sample items to bloggers to publicize your product. You want to gift the bloggers with the largest readership for the gifts, with the least overlap.

3.5 Graph Programming Tools

As of 2012, there are no ANSI or ISO standard graph query languages. We have to depend on proprietary languages or open-source projects. They depend on underlying graph databases, which are also proprietary or open-source projects. Support for some of the open-source projects is available from commercial providers; Neo4j is the most popular product and it went commercial in 2009 after a decade in the open-source world. This has happened in the relational world with PostgreSQL, MySQL, and other products, so we should not be surprised.

There is an ISO standard known as resource description framework (RDF), which is a standard model for data interchange on the Web. It is based on the RDF that extends the linking structure of the Web to use URIs (uniform resource identifiers) to name the relationship between things as well as the two ends of the link (a *triple*). URIs can be classified as locators (URLs), names (URNs), or both. A uniform resource name (URN) functions like a person's name, while a uniform resource locator (URL) resembles that person's street address. In other words, the URN defines an item's identity, while the URL provides a method for finding it.

The differences are easier to explain with an example. The ISBN uniquely identifies a specific edition of a book, its identity. But to read the book, you need its location: a URL address. A typical URL would be a file path for the electronic book saved on a local hard disk.

Since the Web is a huge graph database, many graph databases build on RDF standards. This also makes it easier to have a distributed graph database

that can use existing tools.

3.5.1 Graph Databases

Some graph databases were built on an existing data storage system to get started, but then were moved to custom storage engines. The reason for this is simple: performance. Assume you want to model a simple one-to-many relationship, such as the Kevin Bacon problem. In RDBMS and SQL, there will be a table for the relationship, which will contain a reference to the table with the unique entity and a reference for each row matching to that entity in the many side of the relationship. As the relational tables grow, the time to perform the join increases because you are working with entire sets.

In a graph model, you start at the Kevin Bacon node and traverse the graph looking at the edges with whatever property you want to use (e.g., “was in a movie with”). If there is a node property, you then filter on it (e.g., “this actor is French”). The edges act like very small, local relationship tables, but they give you a traversal and not a join.

A graph database can have ACID transactions. The simplest possible graph is a single node. This would be a record with named values, called properties. In theory, there is no upper limit on the number of properties in a node, but for practical purposes, you will want to distribute the data into multiple nodes, organized with explicit relationships.

3.5.2 Graph Languages

There is no equivalent to SQL in the graph database world. Graph theory was an established branch of mathematics, so the basic terminology and algorithms were well-known were the first products came along. That was an advantage. But graph database had no equivalent to IBM’s System R, the research project that defined SEQUEL, which became SQL. Nor has anyone tried to make one language into the ANSI, ISO or other industry standard.

SPARQL

SPARQL (pronounced “sparkle,” a recursive acronym for SPARQL Protocol and RDF Query Language) is a query language for the RDF format. It tries to look a little like SQL by using common keywords and a bit like C with special ASCII characters and lambda calculus. For example:

```
PREFIX abc: < http://example.com/exampleOntology#>
```

```
SELECT ?capital ?country
WHERE {
  ?x abc:cityname ?capital;
  abc:isCapitalOf ?y.
  ?y abc:countryname ?country;
  abc:isInContinent abc:Africa.}
```

where the ? prefix is a free variable, and : names a source.

SPASQL

SPASQL (pronounced “spackle”) is an extension of the SQL standard, allowing execution of SPARQL queries within SQL statements, typically by treating them as subqueries or function clauses. This also allows SPARQL queries to be issued through “traditional” data access APIs (ODBC, JDBC, OLE DB, ADO.NET, etc.).

Gremlin

Gremlin is an open-source language that is based on traversals of a property graph with a syntax taken from OO and the C programming language family (*https://github.com/tinkerpop/gremlin/wiki*). There is syntax for directed edges and more complex queries that looks more mathematical than SQL-like. Following is a sample program. Vertexes are numbered and a traversal starts at one of them. The path is then constructed by in–out paths on the ‘likes’ property:

```
g = new Neo4jGraph('/tmp/neo4j')
// calculate basic collaborative filtering for vertex 1
m = [:]
g.v(1).out('likes').in('likes').out('likes').groupCount(m)
m.sort{-it.value}
// calculate the primary eigenvector (eigenvector centrality) of
  a graph
m = [:]; c = 0;
g.V.out.groupCount(m).loop(2){c++ < 1000}
m.sort{-it.value}
```

Eigenvector centrality is a measure of the influence of a node in a network.

It assigns relative scores to all nodes in the network based on the concept that connections to high-scoring nodes contribute more to the score of the node in question than equal connections to low-scoring nodes. It measures the effect of the “cool kids” in your friends list. Google’s PageRank is a variant of the eigenvector centrality measure.

Cypher (NEO4j)

Cypher is a declarative graph query language that is still growing and maturing, which will make SQL programmers comfortable. It is not a weird mix of odd ASCII charterers, but human-readable keywords in the major clauses. Most of the keywords like `WHERE` and `ORDER BY` are inspired by SQL. Pattern matching borrows expression approaches from SPARQL. The query language is comprised of several distinct clauses:

- `START`: starting points in the graph, obtained via index lookups or by element IDs.
- `MATCH`: the graph pattern to match, bound to the starting points in `START`.
- `WHERE`: filtering criteria.
- `RETURN`: what to return.
- `CREATE`: creates nodes and relationships.
- `DELETE`: removes nodes, relationships, and properties.
- `SET`: sets values to properties.
- `FOREACH`: performs updating actions once per element in a list.
- `WITH`: divides a query into multiple, distinct parts.

For example, following is a query that finds a user called John in an index and then traverses the graph looking for friends of John’s friends (though not his direct friends) before returning both John and any friends-of-friends who are found:

```
START john = node:node_auto_index(name = ‘John’)
MATCH john-[:friend]- > ()-[:friend]- > fof
RETURN john, fof
```

We start the traversal at the `john` node. The `MATCH` clause uses arrows to show the edges that build the friend-of-friend edges into a path. The final clause tells the query what to return.

In the next example, we take a list of users (by node ID) and traverse the graph looking for those other users who have an outgoing friend relationship, returning only those followed users who have a name property starting with S:

```
START user = node(5,4,1,2,3)
MATCH user-[:friend]- > follower
WHERE follower.name = ~ 'S.*'
RETURN user, follower.name
```

The `WHERE` clause is familiar from SQL and other programming languages. It has the usual logical operators of `AND`, `OR`, and `NOT`; comparison operators; simple math; regular expressions; and so forth.

Trends

Go to *http://www.graph-database.org/* for PowerPoint shows on various graph language projects. It will be in flux for the next several years, but you will see several trends. The proposed languages are declarative, and are borrowing ideas from SQL and the RDBMS model. For example, GQL (Graph Query Language) has syntax for `SUBGRAPH` as a graph venison of a derived table. Much like SQL, the graph languages have to send data to external users, but they lack a standard way of handing off the information.

It is probably worth the effort to get an open-source download of a declarative graph query language and get ready to update your resume.

Concluding Thoughts

Graph databases require a change in the mindset from computational data to relationships. If you are going to work with one of these products, then you ought to get math books on graph theory. A short list of good introductory books are listed in the Reference section.

References

1. Celko J. *Trees and hierarchies in SQL for smarties*. Burlington, MA: Morgan-Kaufmann; 2012.
2. Chartrand G. *Introductory graph theory*. Mineola, NY: Dover Publications; 1984.
3. Chartrand G, Zhang P. *A first course in graph theory*. New York:

McGraw-Hill; 2004.

4. Gould R. *Graph theory*. Mineola, NY: Dover Publications; 2004.
5. Trudeau RJ. *Introduction to graph theory*. Mineola, NY: Dover Publications; 1994.
6. Maier D. *Theory of relational databases*. Rockville. MD: Computer Science Press; 1983.
7. Wald A. *Sequential analysis*. Mineola, NY: Dover Publications; 1973.

CHAPTER 4

MapReduce Model

Abstract

The MapReduce model was developed by Google and Yahoo for their internal use. Google created the Hadoop distributed file system and Yahoo developed Pig Latin to handle their volume of data. These products became open source. Hadoop dominates the NoSQL market as part of the SMAQ stack, the NoSQL counterpart of the LAMP stack for websites. The process has two phases: mapping and reducing. The mapping phase gets the data in a parallelized fashion. The reduce phase filters and aggregates this data to produce a final result.

Keywords

ETL (extract transform load); Google; Hadoop; HDFS (Hadoop distributed file system); LAMP stack; MapReduce; Pig Latin; RAID storage systems; SMAQ stack; Yahoo

Introduction

This chapter discusses the MapReduce model of data processing developed by Google and Yahoo for their internal use. This is a data retrieval model rather than a query model.

The Internet as we know it today, or Web 2.0 if you prefer, really started with the LAMP stack of open-source software that anyone could use to get up a website: Linux (operating system), Apache (HTTP server), MySQL (database, but since it was acquired by Oracle, people are moving to the open-source version, MariaDB), and PHP, Perl, or Python for the application language. Apache and MySQL are now controlled by Oracle Corporation and the open-source community is distrustful of them.

There is a similar stack in the Big Data storage world called the SMAQ stack for storage, MapReduce, and query, rather than particular products per se. Like the LAMP stack, the tools that implement the layers of the SMAQ are usually open source and run on commodity hardware. The operative word here is “commodity” so that more shops can move to Big Data models.

This leads to the obvious question as to what Big Data is. The best answer I

found is when the size of the data involved in the project is a major concern for whatever reason. We are looking for projects driven by the data, not computations or analysis of the data. The first web applications that hit this problem were web search engines. This makes sense; they were trying to keep up with the growth of the Web and not drop or leave out anything.

Today, there are other players on the Web with size problems. The obvious ones are social networks, multiplayer games and simulations, as well as large retailers and auction sites. But outside of the Web, mobile phones, sensors, and other constant data flows can create petabytes of data.

Google invented the basic MapReduce technique, but Yahoo actually turned it into the Hadoop storage tools. As of this writing. Hadoop-based systems have a majority of the storage architectures. The query part can be done with Java because Hadoop is written in Java, but there are higher-level query languages for these platforms (more on that later).

The MapReduce part is the heart of this model. Imagine a large open office with clerks sitting at their desks (commodity hardware), with piles of catalogs in front of them. Putting the catalogs on their desks is a batch process; it is not like an SQL transaction model with interactive insert, update, and delete actions on the database.

Keeping with the office clerks image, once a day (or whatever temporal unit), the mail clerks dump the day's catalogs on the clerk's desks. What the clerks do not see is that the mail room (data sources) has to be cleaned up, filtered, and sorted a bit before the mail gets put in the mail cart and distributed. The ETL (extract, transform, load) tools from data warehousing work in Big Data, too, but the data sources are not often the clean, traditional structured ones that commercial data warehouses use. That is a whole topic in itself.

But assume we are ready for business. A boss at the front of the room shouts out the query: "Hey, find me a pair of red ballet flats!" to everyone, at the same time. Some of the clerks might realize that they do not have shoe catalogs in the pile on their desk and will ignore the request. The rest of the clerks will snap to attention and start looking through their catalogs. But what are they using as a match? A human being knows that we asked for a particular kind and color of women's shoes. A human being will look at a picture and understand it. A computer has to be programmed to do this, and that might include a weighted match score and not a yes/no result. The smarter the algorithm, the longer it takes to run, and the more it costs in resources.

This is the mapping part. The query has to be parallelized. In this analogy, shouting out a query is enough, but the real world is not that simple. You have to have tasks that can be done independently of each other and yet consolidated into an alpha result. Another mail clerk has to run down the rows of desks and pick up the hits from the clerks, as they finish at different rates. Some clerks will have no matches and we can skip them. Some clerks will have an exact match to "red ballet flats" in their catalog; some clerks will have "ballet flats" or "red flats" near-matches.

Now it is time for the reduce phase. The mail clerk gets the catalog clerks' notes to the boss at the front of the room. But their handwriting is sloppy, so the mail clerk has to summarize and sort these notes. More algorithms, and a presentation layer now! Finally the boss has his or her answer and we are ready for another query.

Notice that this is more of a retrieval than what an SQL programmer would think of as a query. It is not elaborate like a relational division, roll up, cube, or other typical aggregation in SQL. This leads us to the storage used and finally the query languages

4.1 Hadoop Distributed File System

The standard storage mechanism used by Hadoop is the Hadoop distributed file system (HDFS). It is built from commodity hardware arranged to be fault tolerant. The nature of commodity hardware is that when we have a failure, the bad unit can be swapped out. This is the reason that RAID storage works. But we want extreme scalability, up to petabytes. This is more data than the usual RAID storage system handles.

The next assumption is that it will be streaming data rather than random data access. The data is just stuffed into disks while RAID systems have deliberate redundancy in the data that has to be controlled by the RAID system. This is a write-once model that assumes data never changes after it is written. This model simplifies replication and speeds up data throughput. But it means that the front end has to do any validation and integrity checking before the data gets into the system.

RDBMS people hate this lack of data integrity. We want `CHECK()` constraints and referential integrity enforced by `FOREIGN KEY` constraints in the database. It is a file system, not a database. The Big Data model is that we might get data integrity eventually. In the meantime, we assume that we can live with some level of incorrect and missing data.

HDFS is portable across operating systems, but you will find that LINUX is the most popular platform. This should be no surprise, since it was so well established on the Web.

The huge data volume makes it is much faster to move the program near to the data, and HDFS has features to facilitate this. HDFS provides an interface similar to that of regular file systems. Unlike a database, HDFS can only store and retrieve data, not index it. Simple random access to data is not possible.

4.2 Query Languages

While it is possible to use a native API to get to the HDFS, developers prefer a higher-level interface. They are faster to code, they document the process, and the code can port to another platform or compiler.

4.2.1 Pig Latin

Pig Latin, or simply Pig, was developed by Yahoo and is now part of the Hadoop project. It is aimed at developers who use a workflow or directed graph programming model. That model can be parallelized, but each path has to be executed in order.

The typical Pig program has a `LOAD` command at the front and a `STORE` command at the end. Another characteristic that is not quite like procedural programming is that assignments are permanent; you cannot change a name. Unfortunately, you can reuse it without losing the prior object. Think of each statement as a station in an assembly line. You fetch a record from a source, pass it to the next station, and fetch the next record. The next station will do its task with whatever data it has.

For example, the fields in a record are referenced by a position using a dollar sign and a number, starting with $0. Following is the example Pig program from the Wikipedia article on the language. It extracts words by pulling lines from text and filtering out the whitespace. The data is grouped by words, each group is counted, and the final counts go to a file:

```
input_lines = LOAD '/tmp/my-copy-of-all-pages-on-internet' AS
  (line:chararray);

-- Extract words from each line and put them into a pig bag

-- datatype, then flatten the bag to get Alpha word on each row

words = FOREACH input_lines GENERATE FLATTEN(TOKENIZE(line)) AS
  word;
```

```
— FILTER out any words that are just white spaces

filtered_words = FILTER words BY word MATCHES '\w + ';

— create a GROUP for each word

word_Groups = GROUP filtered_words BY word;

— count the entries in each GROUP

word_count = FOREACH word_Groups GENERATE COUNT(filtered_words)
  AS count, GROUP AS word;

— order the records BY count

ordered_word_count = ORDER word_count BY count DESC;

STORE ordered_word_count INTO '/tmp/number-of-words-on-
  internet';
```

Developers can write user-defined functions (UDFs) in Java to get more expressive power. While SQL also has a `CREATE` function that can use external languages, SQL is expressive and powerful enough that this is seldom done by good SQL programmers.

The `LOAD` command is a cross between the `SELECT` command and the Data Declaration Language (DDL) of SQL. It not only fetches records, but has extra clauses that format the data into something the next statements can handle. The `USING` clause invokes a library procedure and uses it to read from the data source. The `AS (< field pair list >)` clause will break the records into fields and assign a data type to them. The elements in the field pair list are a pair of (< field name >:< data type >) separated by commas. There are also options for structured data of various kinds. `DUMP` will show you the current content of an object for debugging; it does not send it to persistent storage. Do not worry about it until you need to look at your code.

The `FILTER` command is the Pig version of the SQL search condition. It uses a mix of SQL and C symbols, logical operators, and semantics as SQL. For example:

```
Users_20 = FILTER Users BY age > 19 AND age < 30
```

Predicates in the `BY` clause can contain the C equality operators `==` and `! =` rather than the SQL `<>`. The rest of the theta operators are `>`, `>= ` , `<`, and `< =`. These comparators can be used on any scalar data type, and `==` and `! =` can also be applied to maps and tuples. To use these with beta tuples, both tuples must have the same schema or both not have a schema. None of the equality operators can be applied to bags. Pig has the usual operator

precedence and basic math functions. But it is not a computational language, so you do not get a large math function library as part of the language.

Strings are called *chararrays* (array of characters) after the C family model and have Java's regular expression syntax and semantics. Since chararrays are written in Java, this is no surprise, but they can throw off an SQL or other language programmer. SQL is based on the POSIX regular expressions, which have lots of shorthands. PERL regular expressions will work on a portion of a string, while Java does not. For example, if you are looking for all fields that contain the string "Celko" you must use `'.*Celko.*'` and not `'Celko'`, which is an exact match. Only SQL uses `'%'` and `'_ '` for wildcards; everyone else has `'.'` for single characters and `'*'` for a varying-length match.

The usual logical `AND`, `OR`, and `NOT` operators are here with the standard precedence. Pig has SQL's three-valued logic and `NULL`s, so an `UNKNOWN` will be treated as a `FALSE` in a `FILTER`. Pig will short-circuit logical operations when possible. That means a Pig program is executed from left to right, and when the value of a predicate will not be affected by the following terms in the `FILTER`, evaluation stops.

Since Pig allows UDFs and is not a functional language, it means that some code might not execute and have expected side effects. As a silly example, consider a UDF that returns `TRUE`, but not before it has done something outside the application, like reformat all the hard drives:

```
FILTER Foobar BY (1 == 2) AND Format_All_Drives_Function (x);
```

The first term `(1 == 2)` is `FALSE` so `Format_All_Drives_Function (x)` will never be invoked. But if we write it as

```
FILTER Foobar BY Format_All_Drives_Function (x) AND (1 == 2);
```

`Format_All_Drives_Function (x)` is invoked now. The system will disappear if nothing stops this piece of insanity.

Side effects also prevent other optimizations that would be possible if we could be sure there were no side effects. SQL/PSM gets around this by requiring a procedure or function be declared as not deterministic. A deterministic function will return the same output for the same inputs. Think of a mathematical function, like sin() or cos(); now think of a `FETCH` statement that gets whatever the next record in a file happens to be.

There is a website with procedures in Java for all kinds of things. It was too cute to resist, so this website is the Piggybank. The packages are based on the type of function. The current top-level packages correspond to the function

type and are:

- *org.apache.pig.piggybank.comparison*—for a custom comparator used by the `ORDER` operator.
- *org.apache.pig.piggybank.evaluation*—for evaluation functions like aggregates and column transformations.
- *org.apache.pig.piggybank.filtering*—for functions used in the `FILTER` operator.
- *org.apache.pig.piggybank.grouping*—for grouping functions.
- *org.apache.pig.piggybank.storage*—for `LOAD`/`STORE` functions.

The `FOREACH` statement applies an operation to every record in the data pipeline. Unix programmers who have written piped commands will be familiar with this model. `FOREACH` inputs a record named Alpha and outputs an Alpha record to send down the pipeline to the next statement. The next statement will create a new record named Beta. For the RDBMS people, this is (sort of) how Pig implements the relational projection operator. For example, the following code loads an entire record, but then removes all but the user and ID fields from each record:

```
Alpha = LOAD 'input' as (user_name:chararray, user_id:long,
  address:chararray, phone_nbr:chararray);

Beta = FOREACH Alpha GENERATE user_name, user_id;
```

But this is not quite projection in the relational sense. RDBMS is set-oriented, so the projection occurs all at once. Pig is a workflow model—we get a flow of generated tuples as output. Subtle, but important.

`FOREACH` has a lot of tools, the simplest of which are constants and field references. Field references can be by name (the SQL model) or by position (record-oriented model). Positional references are proceeded by a `$` and start from 0. I strongly recommend against the positional references since positions do not document the process. Positions will change if the data source changes. Referencing a nonexistent positional field in the tuple will return `NULL`. Referencing a field name that does not exist in the tuple will produce an error.

Use descriptive names and not positional numbers for safety and as the best modern programming practice. Today, we have text editors and can cut and paste lists and we do not use 80-column punch cards. For example:

```
Prices = LOAD 'NYSE_Daily_Ticker' as (exchange, symbol, date,
  open, high, low, close, volume, adj_close);
```

```
Gain = FOREACH Prices GENERATE close – open; — simple math

—Gain2 = FOREACH Prices GENERATE $6 - $3;
```

`Gain` and `Gain2` will contain the same values. In addition to using names and positions, you can refer to all fields in a record using an `*` like the SQL convention. This produces a tuple that contains all the fields. Beginning in version 0.9, you can also refer to ranges of fields using beta periods and the syntax `[< start field >]..[< end field >]` . The syntax expands the fields in the order declared. If there is no explicit starting column, then the first alpha is used; if there is no explicit ending column, then the last alpha is used; otherwise, the range includes all fields between the start field and end field based on the field list.

The very useful question mark operator, with the syntax `< predicate > ? < true value >: < false value >`, will be familiar to older C programmers. It is the Pig version of the `CASE` expression ancestor! The predicate is tested and the expression returns the true value after the question mark if it is `TRUE`. It returns the false value if the predicate is `FALSE`. This is how the C question mark works in the Boolean logic of its parent language. But Pig has a `NULL`! Quasi-SQL comes into play! Perhaps it is easier to see, as follows:

```
2 == 2 ? 1 : 4 —returns 1

2 == 3 ? 1 : 4 —returns 4

NULL == 2 ? 1 : 4 — returns NULL

2 == 2 ? 1 : ‘Celko’ — type error, string vs integer
```

Pig has bags, which is what SQL is based on. It is a collection of tuples that have no ordering, and allow duplicates in the collection. But SQL—good SQL—will have a `PRIMARY KEY` that assures the bag is actually a real set.

The `GROUP` statement is not the `GROUP BY` statement used by SQL! The `GROUP BY` in SQL is a summarizing statement that returns a table. The Pig `GROUP` statement collects records with the same key together. The result is not a summary, but the intermediate step of building a collection of bags. In Pig, you can apply the aggregate functions if you wish. For example:

```
Daily_Ticker = LOAD ‘Daily_Stock_Prices’ AS (stock_sym,
  stock_price);

Daily_Stock_Groups = GROUP Daily_Ticker BY stock_sym;

Ticker_Cnt = FOREACH Daily_Stock_Groups GENERATE GROUP,
```

```
COUNT(Daily_Ticker);
```

This example groups records by the stock's ticker symbol and then counts them. The records coming out of the `GROUP BY` statement have beta fields, the key, and the bag of collected records. The key field is named `GROUP` and the bag is named for the alias that was grouped. So in the previous examples it will be named `Daily_Ticker` and inherit the schema from `Daily_Ticker`. If the relation `Daily_Ticker` has no schema, then the bag `Daily_Ticker` will have no schema. For each record in the `GROUP`, the entire record, *including the key*, is in the bag.

You can also use `GROUP` on multiple keys. The keys must be in parenthesis, just like an SQL row constructor. But unlike SQL, we can use tuples as fields in the results; so, we still have records with two fields, but the fields are more complicated than SQL's scalar columns.

At this point, it is easier to show this with an example. Let's make up two data sets, `Alpha` and `Beta`:

```
Alpha = LOAD 'Alpha' USING PigStorage();

Beta = LOAD 'Beta' USING PigStorage();
```

PigStorage is a standard library routine that will let us read in records from a standard source. Assume the data looks like this:

```
Alpha:
a   A   1
b   B   2
c   C   3
a   AA   11
a   AAA  111
b   BB   22
Beta:
x   X   a
y   Y   b
x   XX   b
z   Z   c
```

Now we can use some of the fancy commands that resemble their relational cousins. We have already discussed `GENERATE`, but here is how it is dumped. Pay attention to the parentheses and the use of a zero initial position:

```
Alpha_0_2 = FOREACH Alpha GENERATE $0, $2;
(a, 1)
(b, 2)
(c, 3)
(a, 11)
(a, 111)
(b, 22)
```

The `GROUP` statement will display the grouping key first, then the tuple of rows appears in curvy brackets. Math majors will be delighted with this because the curvy bracket is the standard notation for an enumerated set. Notice also that the fields are in their original order:

```
Alpha_Grp_0 = GROUP Alpha BY $0;
(a, {(a, A, 1), (a, AA, 11), (a, AAA, 111)})
(b, {(b, B, 2), (b, BB, 22)})
(c, {(c, C, 3)})
```

When the grouping key is more than one field, the row constructor is in parentheses, but the curvy brackets are still a list of fields:

```
Alpha_Grp_0_1 = GROUP Alpha BY ($0, $1);
((a, A), {(a, A, 1)})
((a, AA), {(a, AA, 11)})
((a, AAA), {(a, AAA, 111)})
((b, B), {(b, B, 2)})
((b, BB), {(b, BB, 22)})
((c, C), {(c, C, 3)})
```

Pig has three basic aggregate functions that look like their SQL cousins: `SUM()`, `COUNT()`, and `AVG()`. The rounding and presentation rules are not unexpected, but Pig does not have all of the fancy SQL operators that have been added to the ANSI/ISO standard SQL over the years. For example:

```
Alpha_Grp_0_Sum = FOREACH Alpha_Grp_0 GENERATE GROUP,
  SUM(Alpha.$2);
(a, 123.0)
(b, 24.0)
(c, 3.0)
```

```
Alpha_Grp_0_Cnt = FOREACH Alpha_Grp_0 GENERATE GROUP,
  COUNT(Alpha);
(a, 3)
(b, 2)
(c, 1)
Alpha_Grp_0_Avg = FOREACH Alpha_Grp_0 GENERATE GROUP, AVG(Alpha);
(a, 41.0)
(b, 12.0)
(c, 3.0)
```

Now we get into the fancy stuff! `FLATTEN` will look familiar to LISP programmers, which has a common function of the same name. It takes the tuples of the curvy brackets and puts them into a list. This is why having the key in the tuples is important; you do not destroy information. For example:

```
Alpha_Grp_0_Flat = FOREACH Alpha_Grp_0 GENERATE FLATTEN(Alpha);
(a, A, 1)
(a, AA, 11)
(a, AAA, 111)
(b, B, 2)
(b, BB, 22)
(c, C, 3)
```

The `COGROUP` is a sort of join. You wind up with three or more fields. The first is what value is common to the tuples that follow. Each of the `BY` clauses tells you which column in the tuple is used. `NULL`s are treated as equal, just as we did in SQL's grouping operators. For example:

```
Alpha_Beta_Cogrp = COGROUP Alpha BY $0, Beta BY $2;
(a, {(a, A, 1), (a, AA, 11), (a, AAA, 111)}, {(x, X, a)})
(b, {(b, B, 2), (b, BB, 22)}, {(y, Y, b), (x, XX, b)})
(c, {(c, C, 3)}, {(z, Z, c)})
```

Again, notice that this is a nested structure. The style in Pig is to build a chain of steps so that the engine can take advantage of parallelism in the workflow model. But that can often mean un-nesting these structures. Look at this example and study it:

```
Alpha_Beta_Cogrp_Flat = FOREACH Alpha_Beta_Cogroup GENERATE
  FLATTEN(Alpha.($0, $2)), FLATTEN(Beta.$1);
```

```
(a, 1, X)
(a, 11, X)
(a, 111, X)
(b, 2, Y)
(b, 22, Y)
(b, 2, XX)
(b, 22, XX)
(c, 3, Z)
```

`JOIN` is the classic relational natural equijoin, but where SQL would drop one of the redundant join columns from the result table, Pig keeps both. This example has the join fields at the ends of the rows, so you can see them. Also notice how `Alpha` and `Beta` retain their identity, so the `$` position notation does not apply to the result:

```
Alpha_Beta_Join = JOIN Alpha BY $0, Beta BY $2;
(a, A, 1, x, X, a)
(a, AA, 11, x, X, a)
(a, AAA, 111, x, X, a)
(b, B, 2, y, Y, b)
(b, BB, 22, y, Y, b)
(b, B, 2, x, XX, b)
(b, BB, 22, x, XX, b)
(c, C, 3, z, Z, c)
```

`CROSS` is the classic relational cross-join or Cartesian product if you prefer classic set theory. This can be dangerous for SQL programmers. In SQL, the `SELECT .. FROM..` statement is defined as a cross-join in the `FROM` clause, and projection in the `SELECT` clause. No SQL engine actually does it this way in the real world, but since Pig is a step-by-step language, you can do exactly that! Essentially, the Pig programmer has to be his or her own optimizer. For example:

```
Alpha_Beta_Cross = CROSS Alpha, Beta;
(a, AA, 11, z, Z, c)
(a, AA, 11, x, XX, b)
(a, AA, 11, y, Y, b)
(a, AA, 11, x, X, a)
```

```
(c, C, 3, z, Z, c)
(c, C, 3, x, XX, b)
(c, C, 3, y, Y, b)
(c, C, 3, x, X, a)
(b, BB, 22, z, Z, c)
(b, BB, 22, x, XX, b)
(b, BB, 22, y, Y, b)
(b, BB, 22, x, X, a)
(a, AAA, 111, x, XX, b)
(b, B, 2, x, XX, b)
(a, AAA, 111, z, Z, c)
(b, B, 2, z, Z, c)
(a, AAA, 111, y, Y, b)
(b, B, 2, y, Y, b)
(b, B, 2, x, X, a)
(a, AAA, 111, x, X, a)
(a, A, 1, z, Z, c)
(a, A, 1, x, XX, b)
(a, A, 1, y, Y, b)
(a, A, 1, x, X, a)
```

Split was in Dr. Codd's original relational operators. It never caught on because it returns two tables and can be done with other relational operators Maier, 1983, pp. 37–38). But Pig has a version of it that lets you split the data into several different "buckets" in one statement, as follows:

```
SPLIT Alpha INTO Alpha_Under IF $2 < 10, Alpha_Over IF $2 > = 10;
— Alpha_Under:
(a, A, 1)
(b, B, 2)
(c, C, 3)
— Alpha_Over:
(a, AA, 11)
(a, AAA, 111)
(b, BB, 22)
```

Did you notice that you could have rows that do not fall into a bucket?

There is a trade-off in this model. In SQL, the optimizer has statistics and uses that knowledge to create an execution plan. If the stats change, the execution plan can change. There is no way to collect statistics in Pig or any web-based environment. Once the program is written, you have to live with it.

This also means there is no way to distribute the workload evenly over the "reducers" in the system. If one of them has a huge workload, everyone has to wait until everyone is ready to pass data to the next step in the workflow. In fact, it might be impossible for one reducer to manage that much data.

Hadoop has a "combiner phase" that does not remove all skew data, but it places a bound on it. And since, in most jobs, the number of mappers will be at most in the tens of thousands, even if the reducers get a skewed number of records, the absolute number of records per reducer will be small enough that the reducers can handle them quickly.

Some calculations like `SUM` that can be decomposed into any number of steps are called *distributive* and they work with the combiner. Remember your high school algebra? This is the distributive property and we like it.

Calculations that can be decomposed into an initial step, any number of intermediate steps, and a final step are called *algebraic*. Distributive calculations are a special case of algebraic, where the initial, intermediate, and final steps are all the same. `COUNT` is an example of such a function, where the initial step is a count and the intermediate and final steps are sums (more counting) of the individual counts. The median is not algebraic. You must have all the records sorted by some field(s) before you can find the middle value.

The real work in Pig is building UDFs that use the combiner whenever possible, because of its skew-reducing features and because early aggregation greatly reduces the amount of data shipped over the network and written to disk, thus speeding up performance significantly. This is not easy, so I am not even going to try to cover it.

4.2.2 Hive and Other Tools

Just as Pig was a Yahoo project, Hive is an open-source Hadoop language from Facebook. It is closer to SQL than Pig and can be used for ad-hoc queries without being compiled like Pig. It is the representative product in a family that includes Cassandra and Hypertable. They use HDFS as a storage

system, but use a table-based abstraction over HDFS, so it is easy to load structured data. Hive QL is the SQL-like query language that executes MapReduce jobs. But it can use the Sqoop to import data from relational databases into Hadoop. It was developed by Cloudera for their Hadoop platform products. Sqoop is database-agnostic, as it uses the Java JDBC database API. Tables can be imported either wholesale, or using queries to restrict the data import. Sqoop also offers the ability to reinject the results of MapReduce from HDFS back into a relational database. This means that Hive is used for analysis and not for online transaction processing (OLTP) or batch processing.

You declare tables with columns as in SQL, using a simple set of data types: `INT`, `FLOAT`, `DATE`, `STRING`, and `BOOLEAN`. The real strength comes from also using simple data structures:

- *Structs:* The elements within the type can be accessed using the dot (`.`) notation. For example, for a column *c* of type `STRUCT {a INT; b INT}` the `a` field is accessed by the expression `c.a`.
- *Maps (key–value tuples):* The elements are accessed using `['element name']` notation. For example, in a map *M* comprising of a mapping from `'group'` → `gid`, the gid value can be accessed using `M ['group']`.
- *Arrays (indexable one-dimensional lists):* The elements in the array have to be in the same type. Elements can be accessed using the [*n*] notation where *n* is an index (zero-based) into the array. For example, for an array A having the elements `['a', 'b', 'c']`, A[1] returns `'b'`:

```
CREATE TABLE Foo
(something_string STRING,
  something_float FLOAT,l
  my_array ARRAY < MAP < ['foobar'],
  STRUCT < p1:INT, p2:INT>>);
SELECT something_string, something_float,
    my_array[0], ['foobar'].p1
  FROM Foo;
```

The tables default to text fields separated by a control-A token and records separated by a new-line token. You can add more clauses to define the deliminators and file layout. For example, a simple CSV file can be defined by adding the following to the end of `CREATE TABLE`:

```
ROW FORMAT DELIMITED
  FIELDS TERMINATED BY '\,'
  STORED AS TEXTFILE
```

Another important part of the declarations is that a table can be partitioned over storage and you manipulate it by the partitions. For example, you can get a clause to get random samples from these partitions:

```
TABLESAMPLE(BUCKET x OUT OF y)
```

The `SELECT` can also use infixed joins, but it allows only equijoins. You have `INNER JOIN`, `LEFT OUTER`, `RIGHT OUTER`, and `FULL OUTER` syntax.

There is also a `LEFT SEMI JOIN` to check if there is a reference in one table to another; this cannot be done decoratively in the SQL subset used. The compiler does not do the same optimizations you are used to in SQL, so put the largest table on the rightmost side of the join to get the best performance. There is also the usual `UNION ALL`, but none of the other set operators.

The usual SQL aggregate functions are used. The other built-in functions are a mix of SQL and C family syntax often with both options—that is, `UPPER()` and `UCASE` are the same function with two names, and so forth.

Concluding Thoughts

If you can read an execution plan in the SQL you know, you should not have any trouble with the basic concepts in the MapReduce model of data. Your learning curve will come with having to use Java and other lower-level tools that are part of an SQL compiler for complicated tasks.

The lack of an SQL style optimizer with statistics, transaction levels, and built-in data integrity will be a major jump. You will find that you have to do manually those things that you have had done for you.

Older programming languages have idioms and conventions, just like human languages. A COBOL programmer will solve a problem in a different way than an SQL programmer would approach the same problem. MapReduce is still developing its idioms and conventions. In fact, it is still trying to find a standard language to match the position that SQL has in RDBMS.

References

1. Maier D. *Theory of relational databases*. Rockville. F. MD:

Computer Science Press; 1983.

2. Capriolo E, Wampler D. *Programming hive*. Cambridge: O’Reilly Media; 2012.
3. Gates A. *Programming Pig*. Cambridge, MA: O’Reilly Media; 2012.

CHAPTER 5

Streaming Databases and Complex Events

Abstract

Streaming databases use traditional structured data, but add a temporal dimension to it. If RDBMS is static data and static processing, then streaming data is moving data and active processing. It has to use a generational concurrency model to catch the flow of data. These databases do not want to respond to a query; the database is taking action. The database monitors a system or process by looking for exceptional behavior and generates alerts when such behavior occurs. When an event is observed, the system has to deliver the right information to the right consumer at the right granularity at the right time. It is personalized information delivery. From this, the consumer can decide on some dynamic operational behavior.

Keywords

dynamic operational behavior; CEP (complex event processing); generational concurrency model; information dissemination; isolation levels in optimistic concurrency; observation; speed; velocity; streaming database; Q language

Introduction

This chapter discusses streaming databases. These tools are concerned with data that is moving through time and has to be “caught” as it flows in the system. Rather than static data, we look for patterns of events in this temporal flow. Again, this is not a computational model for data.

The relational model assumes that the tables are static during a query and that the result is also a static table. If you want a mental picture, think of a reservoir of data that is used as it stands at the time of the query or other Data Manipulation Language (DML) statement. This is, roughly, the traditional RDBMS.

But streaming databases are built on constantly flowing data; think of a river or a pipe of data. The best-known examples of streaming data are stock and commodity trading done by software in subsecond trades. The reason that these applications are well known is because when they mess up, it appears in

the newspapers. To continue the water/data analogy, everyone knows when a pipe breaks or a toilet backs up.

A traditional RDBMS, like a static reservoir, is concerned with data volume. Vendors talk about the number of terabytes (or petabytes or exabytes these days) their product can handle. Transaction volume and response time are secondary. They advertise and benchmark under the assumption that a traditional OLTP system can be made "good enough" with existing technology.

A streaming database, like a fire hose, is concerned with data volume, but more important are *flow rate*, *speed*, and *velocity*. There is a fundamental difference between a bathtub that fills at the rate of 10 liters per minute versus a drain pipe that flows at the rate of 100 liters per minute. There is also a difference between the wide pipe and the narrow nozzle of an industrial pressure cutter at the rate of 100 liters per second.

5.1 Generational Concurrency Models

Databases have to support concurrent users. Sharing data is a major reason to use DBMS; a single user is better off with a file system. But this means we need some kind of "traffic control" in the system.

This concept came from operating systems that had to manage hardware. Two jobs should not be writing to the same printer at the same time because it would produce a garbage printout. Reading data from an RDBMS is not a big problem, but when two sessions want to update, delete, or insert data in the same table, it is not so easy.

5.1.1 Optimistic Concurrency

Optimistic concurrency control assumes that conflicts are exceptional and we have to handle them *after* the fact. The model for optimistic concurrency is microfilm! Most database people today have not even seen microfilm, so, if you have not, you might want to Google it. This approach predates databases by decades. It was implemented manually in the central records department of companies when they started storing data on microfilm. A user did not get the microfilm, but instead the records manager made a timestamped photocopy for him. The user took the copy to his desk, marked it up, and returned it to the central records department. The central records clerk timestamped the updated document, photographed it, and added it to the end of the roll of microfilm.

But what if a second user, user B, also went to the central records department and got a timestamped photocopy of the same document? The central records clerk had to look at both timestamps and make a decision. If user A attempted to put his updates into the database while user B was still working on her copy, then the clerk had to either hold the first copy, wait for the second copy to show up, or return the second copy to user A. When both copies were in hand, the clerk stacked the copies on top of each other, held them up to the light, and looked to see if there were any conflicts. If both updates could be made to the database, the clerk did so. If there were conflicts, the clerk must either have rules for resolving the problems or reject both transactions. This represents a kind of row-level locking, done after the fact.

The copy has a timestamp on it; call it t_0 or `start_timestamp`. The changes are committed by adding the new version of the data to the end of the file with a timestamp, t_1. That is unique within the system. Since modern machinery can work with nanoseconds, an actual timestamp and not just a sequential numbering will work. If you want to play with this model, you can get a copy of Borland's Interbase or its open source, Firebird.

5.1.2 Isolation Levels in Optimistic Concurrency

A transaction running on its private copy of the data is never blocked. But this means that at any time, each data item might have multiple versions, created by active and committed transactions.

When transaction T1 is ready to commit, it gets a `commit_timestamp` that is later than any existing `start_timestamp` or `commit_timestamp`. The transaction successfully `COMMIT`s only if no other transaction, say T2, with a `commit_timestamp` in T1's temporal interval [`start_timestamp`, `commit_timestamp`] wrote data that T1 also wrote. Otherwise, T1 will `ROLLBACK`. This first-committer-wins strategy prevents lost updates (phenomenon P4). When T1 `COMMIT`s, its changes become visible to all transactions of which the `start_timestamps` are larger than T1's `commit_timestamp`. This is called *snapshot isolation*, and it has its own problems.

Snapshot isolation is nonserializable because a transaction's reads come at one instant and the writes at another. Assume you have several transactions working on the same data and a constraint that ($x + y > 0$) on the table. Each

transaction that writes a new value for *x* and *y* is expected to maintain the constraint. While T1 and T2 both act properly in isolation with their copy, the constraint fails to hold when you put them together. The possible problems are:

- *A5 (data item constraint violation):* Suppose constraint C is a database constraint between two data items *x* and *y* in the database. There are two anomalies arising from constraint violation.
- *A5A (read skew):* Suppose transaction T1 reads *x*, and then T2 updates *x* and *y* to new values and `COMMIT`s. Now, if T1 reads *y*, it may see an inconsistent state, and therefore produce an inconsistent state as output.
- *A5B (write skew):* Suppose T1 reads *x* and *y*, which are consistent with constraint C, and then T2 reads *x* and *y*, writes *x*, and `COMMIT`s. Then T1 writes *y*. If there were a constraint between *x* and *y*, it might be violated.
- *P2 (fuzzy reads):* This is a degenerate form of A5A, where ($x = y$). More typically, a transaction reads two different but related items (e.g., referential integrity).
- *A5B (write skew):* This could arise from a constraint at a bank, where account balances are allowed to go negative as long as the sum of commonly held balances remains non-negative.

Clearly, neither A5A nor A5B could arise in histories where P2 is precluded, since both A5A and A5B have T2 write a data item that has been previously read by an uncommitted T1. Thus, phenomena A5A and A5B are only useful for distinguishing isolation levels below `REPEATABLE READ` in strength.

The ANSI SQL definition of `REPEATABLE READ`, in its strict interpretation, captures a degenerate form of row constraints, but misses the general concept. To be specific, Locking a table with a transaction level of `REPEATABLE READ` provides protection from Row Constraint Violations but the ANSI SQL definition forbidding anomalies A1 and A2, does not. Snapshot Isolation is even stronger than `READ COMMITTED`.

The important property for here is that you can be reading data at timestamp t_n while changing data at timestamp $t_{(n + k)}$ in parallel. You are drinking from a different part of the stream and can reconstruct a consistent view of the database at any point in the past.

Table 5.1

ANSI SQL Isolation Levels Defined in Terms of the Three Original Phenomena

Isolation Level	P0 (or A0) Dirty Write	P1 (or A1) Dirty Read	P2 (or A2) Fuzzy Read	P3 (or A3) Phantom
READ UNCOMMITTED	Not Possible	Possible	Possible	Possible
READ COMMITTED	Not Possible	Not Possible	Possible	Possible
REPEATABLE READ	Not Possible	Not Possible	Not Possible	Possible
SERIALIZABLE	Not Possible	Not Possible	Not Possible	Not Possible

Table 5.2

Degrees of Consistency and Locking Isolation Levels Defined in Terms of Locks

Consistency Level = Locking Isolation Level	Read locks on Data Items and Predicates (the same unless noted)	Write Locks on Data Items And Predicates (always the same)
Degree 0	None required	Well-formed writes
Degree 1 = locking Read uncommitted	None required	Well-formed writes, Long duration, Write locks
Degree 2 = locking Read committed	Well-formed reads short duration read locks (both)	Well-formed writes,long duration write locks
Cursor stability	Well-formed reads Read locks held on current of cursor Short duration read predicate locks	Well-formed writes, Long duration write locks
Locking Repeatable read	Well-formed reads Long duration data-item Read locks Short duration read Predicate locks	Well-formed writes, Long duration Write locks
Degree 3 = locking serializable	Well-formed reads Long duration Read locks (both)	Well-formed writes, Long duration Write locks

5.2 Complex Event Processing

The other assumption of a traditional RDBMS is that the constraints on the data model are always in effect when you query. All the water is in one place and ready to drink. But the stream data model deals with complex event processing (CEP). That means that not all the data has arrived in the database yet! You cannot yet complete a query because you are anticipating data, but you know you have part of it. The data can be from the same source, or from multiple sources.

The event concept is delightfully explained in an article by Gregor Hohpe (2005). Imagine you are in a coffee shop. Some coffee shops are synchronous: a customer walks up to the counter and asks for coffee and pastry; the person behind the counter puts the pastry into the microwave, prepares the coffee, takes the pastry out of the microwave, takes the payment, gives the tray to the customer, and then turns to serve the next customer. Each customer is a single thread.

Some coffee shops are asynchronous: the person behind the counter takes the order and payment and then moves on to serve the next customer; a short-order cook heats the pastry and a barista makes the coffee. When both coffee and pastry are ready, they call the customer for window pickup or send a server to the table with the order. The customer can sit down, take out a laptop, and write books while waiting.

The model has producers and consumers, or input and output streams. The hot pastry is an output that goes to a consumer who eats it. The events can be modeled in a data flow diagram if you wish.

In both shops, this is routine expected behavior. Now imagine a robber enters the coffee shop and demands people's money. This is not, I hope, routine expected behavior. The victims have to file a police report and will probably call credit card companies to cancel stolen credit cards. These events will trigger further events, such as a court appearance, activation of a new credit card, and so forth.

5.2.1 Terminology for Event Processing

It is always nice to have terminology when we talk, so let's discuss some. A situation is an event that might require a response. In the coffee shop example, running low on paper cups is an event that might not require an immediate response yet. ("How is the cup situation? We're low!") However, running out of paper cups is an event that does require a response since you cannot sell coffee without them. The pattern is detect → decide → respond, either by people or by machine.

1. *Observation:* Event processing is used to monitor a system or process by looking for exceptional behavior and generating alerts when such behavior occurs. In such cases, the reaction, if any, is left to the consumers of the alerts; the job of the event processing application is to produce the alerts only.

2. *Information dissemination:* When an event is observed, the system has to deliver the right information to the right consumer at the right granularity at

the right time. It is personalized information delivery. My favorite is the emails I get from my banks about my checking accounts. One bank lets me set a daily limit and would *only warn me if I went over the limit.* Another bank sends me a short statement of *yesterday's transactions each day.* They do not tell my wife about my spending or depositing, just me.

3. *Dynamic operational behavior:* In this model, the actions of the system are automatically driven by the incoming events. The online trading systems use this model. Unfortunately, a system does not have to have good judgment and needs a way to prevent endless feedback. This situation lead to Amazon's $23,698,655.93 book about flies in 2011. Here is the story: It regards Peter Lawrence's *The Making of a Fly*, a biology book about flies, which published in 1992 and is out of print. But Amazon listed 17 copies for sale: 15 used from $35.54, and 2 new from two legitimate booksellers. The prices kept rising over several days, slowly converging in a pattern. The copy offered by Bordeebook was 1.270589 times the price of the copy offered by Profnath. Once a day Profnath set their price to be 0.9983 times higher than Bordeebook's price. The prices would remain close for several hours, until Bordeebook "noticed" Profnath's change and elevated their price to 1.270589 times Profnath's higher price.

Amazon retailers are increasingly using algorithmic pricing (something Amazon itself does on a large scale), with a number of companies offering pricing algorithms/services to retailers that do have a sanity check. The Profnath algorithm wants to make their price one of the lowest price, but only by a tiny bit so it will sort to the top of the list. A lot of business comes from being in first three positions!

On the other hand, Bordeebook has lots of positive feedback, so their algorithm bets that buyers will pay a premium for that level of confidence. Back in the days when I owned bookstores, there were "book scouts" who earned their living by searching through catalogs and bookstores and buying books for clients. Bordeebook's algorithm seems to be a robot version of the book scout, with a markup of 1.270589 times the price of the book.

4. *Active diagnostics:* The event processing application diagnoses a problem by observing symptoms. A help desk call is the most common example for most of us. There is a classic Internet joke about the world's first help desk (*http://www.why-not.com/jokes.old/archive/1998/january/27.html*) that begins with "This fire help. Me Groog"; "Me Lorto. Help. Fire not work," with the punchline being Grogg beating Lorto with a club for being too stupid to live.

Manufacturing systems look for product failures based on observable symptoms. The immediate goal is to correct flaws in the process but the real goal is to find the root cause of these symptoms. If you view manufacturing as a stream, you will learn about a statistical sampling teaching called *sequential analysis*, invented by Abraham Wald (1973) for munitions testing. (The basic idea is that testing bullets, light bulbs, and many other goods is destructive, so we want to pull the smallest sample size from the stream that will give us the confidence level we want. But the sample size does not have to be constant! If we have a lot of failures, then we want to increase the sample size; if the failure rate is low, we can reduce the test sample size. A technique called *Haldanes' inverse sampling* plays into the adjustment, if you want to do more research.

This idea actually goes back to the early days of statistics under the modern name "gambler's ruin" in the literature. A gambler who raises his bet to a fixed fraction of bankroll when he wins, but does not reduce his bet when he loses, will eventually go broke, even if he has a positive expected value on each bet. There are problems that can only be solved with sequential analysis.

5. *Predictive processing:* You want to identify events before they have happened, so you can eliminate their consequences; in fact, the events might not actually exist! Imagine a fraud detection system in a financial institution that looks for patterns. It will act when it suspects fraud and possibly generate false positives, so further investigation will be required before you can be sure whether a fraud has actually taken place or not. The classic examples in finance are employees who do not take vacations (embezzlers), customers who suddenly apply for lots of credit cards (about to go bankrupt), and changes in purchase patterns (stolen identity).

These different classes of events do not exclude each other, so an application may fall into several of these categories.

5.2.2 Event Processing versus State Change Constraints

The event model is not quite the same as a state transition model for a system. State transitions are integrity checks that assure data changes only according to rules for sequences of procedures, of fixed or variable lifespans, warranties, commercial offers, and bids. You can use the Data Declaration Language (DDL) constraints in SQL to assure the state transition constraints via an auxiliary table. I do not go into much detail here since this is more of an SQL

programming topic.

Such constraints can be modeled as a state-transition diagram to enforce the rules when an entity can be updated only in certain ways. There is an initial state, flow lines that show what are the next legal states, and one or more termination states. The original example was a simple state change diagram of possible marital states that looked like Figure 5.1.

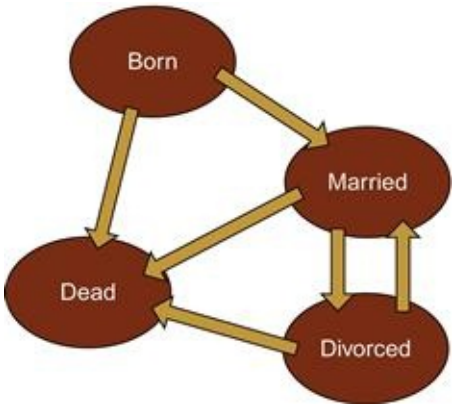


FIGURE 5.1 Martial Status State Transitions.

Here is a skeleton DDL with the needed `FOREIGN KEY` reference to valid state changes and the date that the current state started for me:

```
CREATE TABLE MyLife
(previous_state VARCHAR(10) NOT NULL,
  current_state VARCHAR(10) NOT NULL,
  CONSTRAINT Valid_State_Change
  FOREIGN KEY (previous_state, current_state)
  REFERENCES StateChanges (previous_state, current_state),
  start_date DATE NOT NULL PRIMARY KEY,
  --etc.);
```

These are states of being locked into a rigid pattern. The initial state in this case is “born” and the terminal state is “dead,” a *very* terminal state of being. There is an implied temporal ordering, but no timestamps to pinpoint them in time. An acorn becomes an oak tree before it becomes lumber and finally a chest of drawers. The acorn does not jump immediately to being a chest of drawers. This is a constraint and not an event; there is no action per se.

5.2.3 Event Processing versus Petri Nets

A Petri net is a mathematical model of a system that is handy for CEP. There

is a lot of research on them and they are used to design computer systems with complex timing problems. A Petri net consists of places (shown as circles), transitions (shown as lines or bars), and arcs (directed arrows).

Arcs run from a place to a transition or vice versa, never between places or between transitions. The places from which an arc runs to a transition are called the input places of the transition; the places to which arcs run from a transition are called the output places of the transition.

Graphically, places in a Petri net contain zero or more tokens (shown as black dots). The tokens will move around the Petri when a transition fires. A transition of a Petri net may fire whenever there are sufficient tokens at the start of all input arcs. When a transition fires, it consumes these input tokens, and places new tokens in the places at the end of all output arcs in an atomic step.

Petri nets are nondeterministic: when multiple transitions are enabled at the same time, any one of them may fire. If a transition is enabled, it may fire, but it doesn't have to. Multiple tokens may be present anywhere in the net (even in the same place).

Petri nets can model the concurrent behavior of distributed systems where there is no central control. Fancier versions of this technique have inhibitor arcs, use colored tokens, and so forth. But the important point is that you can prove that a Petri net can be designed to come to a stable state from any initial marking. The classic textbook examples are a two-way traffic light and the dining philosophers problem. You can see animations of these Petri nets at *http://www.informatik.uni-hamburg.de/TGI/PetriNets/introductions/aalst/*.

The dining philosophers problem was due to Edsger Dijkstra as a 1965 student exam question, but Tony Hoare gave us the present version of it. Five philosophers sit at a table with a bowl of rice in front of each of them. A chopstick is placed between each pair of adjacent philosophers. Each philosopher must alternately think and eat. However, a philosopher can only eat rice when he or she has a pair of chopsticks. Each chopstick can be held by only one philosopher and so a philosopher can use a chopstick only if it's not being used by another philosopher. After he or she finishes eating, a philosopher needs to put down both chopsticks so they become available to others. A philosopher can grab the chopstick on his or her right or left as they become available, but cannot start eating before getting both of them. Assume the rice bowls are always filled.

No philosopher has any way of knowing what the others will do. The

problem is how to keep the philosophers forever alternating between eating and thinking and not have anyone starve to death.

5.3 Commercial Products

You can find commercial products from IBM (SPADE), Oracle (Oracle CEP), Microsoft (StreamInsight), and smaller vendors, such as StreamBase (stream-oriented extension of SQL) and Kx (Q language, based on APL), as well as open-source projects (Esper, stream-oriented extension of SQL).

Broadly speaking, the languages are SQL-like and readable, or they are C-like and cryptic. As examples of the two extremes, let's look at StreamBase and Kx.

5.3.1 StreamBase[1]

StreamBase makes direct extensions to a basic version of SQL that will be instantly readable to an SQL programmer. As expected, the keyword `CREATE` adds persistent objects to the schema; here is partial list of DDL statements:

```
CREATE SCHEMA
CREATE TABLE
CREATE INPUT STREAM
CREATE OUTPUT STREAM
CREATE STREAM
CREATE ERROR INPUT STREAM
CREATE ERROR OUTPUT STREAM
```

Notice that streams are declared for input and output and for error handling. Each table can be put in `MEMORY` or `DISK`. Each table must have a `PRIMARY KEY` that that be implemented with a `[USING {HASH | BTREE}]` clause. Secondary indexes are optional and are declared with a `CREATE INDEX` statement.

The data types are more like Java than SQL. There are the expected types of `BOOL`, `BLOB`, `DOUBLE`, `INT`, `LONG`, `STRING`, and `TIMESTAMP`. But they include structured data types (`LIST`, `TUPLE`, `BLOB`) and special types that apply to the StreamBase engine (`CAPTURE`, named schema data type, etc.). The `TIMESTAMP` is the ANSI style date and time data type, and also can be used as an ANSI interval data type. The `STRING` is not a fixed-length data type, as in SQL. `BOOL` can be {`TRUE`, `FALSE`}.

The extensions for streams are logical. For example, we can use a statement to declare dynamic variables that get their values from a stream. Think of a ladle dipping into a stream with the following syntax:

```
DECLARE < variable_identifier > < data
  type > DEFAULT < default_value >
[UPDATE FROM ‘(‘stream_query’)’];
```

In this example, the dynamic variable `dynamic_var` changes value each time a tuple is submitted to the input stream `Dyn_In_Stream`. In the `SELECT` clause that populates the output stream, the dynamic variable is used as an entry in the target list as well as in the `WHERE` clause predicate:

```
CREATE INPUT STREAM Dyn_In_Stream (current_value INT);
DECLARE dynamic_var INT DEFAULT 15 UPDATE FROM
  (SELECT current_value FROM Dyn_In_Stream);
CREATE INPUT STREAM Students
  (student_name STRING, student_address STRING, student_age
    INT);
CREATE OUTPUT STREAM Selected_Students AS
  SELECT *, dynamic_var AS minimum_age FROM Students WHERE
    age > = dynamic_val;
```

One interesting streaming feature is the metronome and heartbeat. It is such a good metaphor. Here is the syntax:

```
CREATE METRONOME  < metronome_identifier >
   (< field_identifier >, < interval >);
```

The `< field_identifier >` is a tuple field that will contain the timestamp value. The `< interval >` is an integer value in seconds. The `METRONOME` delivers output tuples periodically based on the system clock. In the same way that a musician’s metronome can be used to indicate the exact tempo of a piece of music, `METRONOME` can be used to control the timing of downstream operations.

At fixed intervals, a `METRONOME` will produce a tuple with a single timestamp field named `< field_identifier >` that will contain the current time value from the system clock on the computer hosting the StreamBase application. A `METRONOME` begins producing tuples as soon as the application starts.

Its partner is the `HEARTBEAT` statement, with this syntax:

```
< stream expression >
  WITH HEARTBEAT ON < field_identifier >
  EVERY < interval > [SLACK < timeout >]
  INTO < stream_identifier >
  [ERROR INTO < stream_identifier >]
```

The `< field_identifier >` is the field in the input stream that holds a timestamp data type. The `< interval >` is the value in decimal seconds at which the heartbeat emits tuples. The `< timeout >` is the value in decimal seconds that specifies a permissible delay in receiving the data tuple.

`HEARTBEAT` adds timer tuples on the same stream with your data tuples so that downstream operations can occur even if there is a lull in the incoming data. `HEARTBEAT` detects late or missing tuples. Like `METRONOME`, `HEARTBEAT` uses the system clock and emits output tuples periodically, but `HEARTBEAT` can also emit tuples using information in the input stream, independent of the system clock.

`HEARTBEAT` passes input tuples directly through to the output stream, updating its internal clock. If an expected input tuple does not arrive within the configured `< interval >` plus a `< timeout >` value, then `HEARTBEAT` synthesizes a tuple, with all `NULL` data fields except for the timestamp, and emits it.

`HEARTBEAT` sits on a stream and passes through data tuples without modification. The data tuple must include a field of type `< timestamp >`. At configurable intervals, `HEARTBEAT` inserts a tuple with the same schema as the data tuple onto the stream. Fields within tuples that originate from the `HEARTBEAT` are set to `NULL`, with the exception of the `< timestamp >` field, which always has a valid value.

`HEARTBEAT` does not begin to emit tuples until the first data tuple has passed along the stream. `HEARTBEAT` emits a tuple whenever the output `< interval >` elapses on the system clock or whenever the data tuple's `< timestamp >` field crosses a multiple of the output `< interval >`. If a data tuple's `< timestamp >` field has a value greater than the upcoming `HEARTBEAT < interval >`, `HEARTBEAT` immediately emits as many tuples as needed to bring its `< timestamp >` in line with the `< timestamp >` values currently on the stream.

`HEARTBEAT` generates a stream and can be used anywhere a stream expression is acceptable. The following example illustrates the use of

```
HEARTBEAT:
 CREATE INPUT STREAM Input_Trades
 (stock_symbol STRING,
   stock_price DOUBLE,
   trade_date TIMESTAMP);
 CREATE OUTPUT STREAM Output_Trades_Trades;
 CREATE ERROR Output_Trades STREAM Flameout;
```

This can be used with a query, like this:

```
SELECT * FROM Input_Trades
  WITH HEARTBEAT ON trade_date
  EVERY 10.0 SLACK 0.5
  INTO Output_Trades
  ERROR INTO Flameout;
```

There are many more features to work with the streams. The stream can be partitioned into finite blocks of tuples with the `CREATE [MEMORY | DISK] MATERIALIZED WINDOW < window name >`, and these subsets of an infinite stream can be used much like a table. Think of a bucket drawing data from a stream. The `BSORT` can reorder slightly disordered streams by applying a user-defined number of sort passes over a buffer. `BSORT` produces a new, reordered stream. There are other tools, but this is enough for an overview.

While you can write StreamSQL as a text file and compile it, the company provides a graphic tool, StreamBase Studio, that lets you draw a “plumbing diagram” to produce the code. The diagram is also good documentation for a project.

5.3.2 Kx[2]

Q is a proprietary array processing language developed by Arthur Whitney and commercialized by Kx Systems. The Kx products have been in use in the financial industry for over two decades. The language serves as the query language for kdb +, their disk-based/in-memory, columnar database.

The Q language evolved from K, which evolved from APL, short for A Programming Language, developed by Kenneth E. Iverson and associates. It enjoyed a fad with IBM and has special keyboards for the cryptic notation that features mix of Greek, math, and other special symbols.

One of the major problems with APL was the symbols. IBM Selectric typeball could only hold 88 characters and APL has more symbols than that, so you had to use overstrikes. Q, on the other hand, uses the standard ASCII character set.

This family of languages uses a model of atoms, lists, and functions taken in part from LISP. Atoms are indivisible scalars and include numeric, character, and temporal data types. Lists are ordered collections of atoms (or other lists) upon which the higher-level data structures like dictionaries and tables are internally constructed. The operators in the language are functions that use *whole structures* as inputs and outputs. This means that there is no operator precedence; the code is excluded from *right to left*, just like nested function calls in mathematics. Think about how you evaluate sin(cos(*x*)) in math; first compute the cosine of *x*, then apply the sine to that result. The parentheses will become a mess, so we also use the notation f∘g (*x*) for functional composition in math. In Q, they are simply written in sequence.

The data types are the expected ones (given with the SQL equivalents shortly), but the temporal types are more complete than you might see in other languages. The numerics are Boolean (`BIT`), byte (`SMALLINT`), int (`INTEGER`), long (`BIGINT`), real (`FLOAT`), and float (`DOUBLE PRECISION`). The string tapes are char (`CHAR(1)`) and symbol (`VARCHAR(n)`). The temporal data types are date (`DATE`), date time (`TIMESTAMP`), minute (`INTERVAL`), second (`INTERVAL`), and time (`TIME`).

The language also uses the IEEE floating-point standard infinities and NaN values. If you do not know about them, then you need to do some reading. These are special values that represent positive and negative infinities, and thing that are “not a number” with special rules for their use.

There is also an SQL style “select [*p*] [by *p*] from texp [where *p*]” expression. The `by` clause is a change in the usual SQL syntax, best shown with an example. Start with a simple table:

```
tdetails
eid | name iq
--| -----
1001| Dent 98
1002| Beeblebrox 42
1003| Prefect 126
```

The following statement and its results are shown below. The `count..`

`by..` sorts the rows by `iq` and returns the relative position in the table. The max is the highest value of `sc` in the `from` clause table:

```
select topsc:max sc, cnt:count sc by eid.name from tdetails
  where eid.name <> `Prefect

name | topsc cnt

----| ---

Beeblebrox| 42 2

Dent | 98 2
```

Do not be fooled by the SQL-like syntax. This is still a columnar data model, while SQL is row-oriented. The language includes columnar operators. For example, a delete can remove a column from a table to get a new table structure. This would not work in SQL.

Likewise, you will see functional versions of procedural control structures. For example, assignment is typically done with `SET`, `:=`, or `=` in procedural languages, while Q uses a colon. The colon shows that the name on the left side is a name for the expression on the right; it is not an assignment in the procedural sense.

Much like the CASE expression in SQL or ADA, Q has

```
$[expr_cond1; expr_true1; . . . ; expr_condn; expr_truen;
  expr_false]
```

Counted iterations are done with a functional version of the classic `do` loop:

```
do[expr_count; expr_1; . . . ; expr_n]
```

where `expr_count` must evaluate to an int. The expressions `expr_1` through `expr_n` are evaluated `expr_count` times in left-to-right order. The `do` statement does not have an explicit result, so you cannot nest it inside other expressions. Following is factorial *n* done with (*n* − 1) iterations. The loop control is `f` and the argument is `n`:

```
n:5

do[-1 + f:r:n; r*:f-:1]

r

120
```

The conditional iteration is done with a functional version of the classic `while` loop:

```
while[‘expr_cond; expr_1; … ; expr_n]
```

where `expr_ cond` is evaluated and the expressions `expr_1` through `expr_n` are evaluated repeatedly in left-to-right order as long as `expr_cond` is nonzero. The `while` statement does not have an explicit result.

I am going to stop at this point, having given you a taste of Q language coding. It is probably very different from the language you know and this book is not a tutorial for Q. You can find a good tutorial by Jeffry A. Borror entitled “Q for Mortals” at *http://code.kx.com/wiki/JB:QforMortals2/contents*.

The trade-off in learning the Q language is that the programs are insanely fast and compact. An experienced Q programmer can write code rapidly in response to an immediate problem.

Concluding Thoughts

Complex events and streaming data are like the other new technologies in this book. They do not have idioms, conventions, and a standard language. But we are living in a world where the speed of processing is reaching the speed of light. The real problem is having to anticipate how to respond to complex events before they actually happen. There is no time to take action after or during the event.

References

1. Chatfield C. *The analysis of time series: Theory and practice*. Boca Raton: Chapman & Hall; 2003.
2. Hohpe G. *Your coffee shop doesn’t use two-phase commit.* Available at 2005; *http://www.enterpriseintegrationpatterns.com/docs/IEEE_Software_D* 2005.
3. Brockwell PJ, Davis RA. *Time series: Theory and methods, Springer series in statistics*. New York: Springer_Verlag; 2009.
4. Wald A. *Sequential analysis*. Mineola, NY: Dover Publications; 1973.

[1]Disclaimer: I have done a video for StreamBase, available at http://www.streambase.com/webinars/real-time-data-management-for-data-professionals/#axzz2KWao5wxo.

[2]Disclaimer: I have done a video for Kx.

CHAPTER 6

Key–Value Stores

Abstract

The key–value store model uses a key to locate a value of some kind. That value can be traditional data, BLOBs, files, or anything that can be put into computer storage. Likewise, the key can be accessed by hashing, indexing, brute-force scans, or any other appropriate method. This is the most primitive model for data retrieval short of a pile of unorganized data. It is not intended for queries, as all it can do is insert, update, delete, and retrieve (or return a "not found" message). The Berkeley database is the most popular open-source product of this type. Any meaning attached to the values comes from the host program as there is no schema to define anything.

Keywords

BDB (Berkeley database); BLOB; CLOB; key–value store; schema versus no schema

Introduction

A key–value store, also called an associative array, is a collection of pairs, (< key >, < value >), that generalize a simple array. The keys are unique within the collection and can be of any data type that can be tested for equality. This is a form of the MapReduce family, but performance depends on how carefully the keys are designed. Hashing becomes an important technique.

This model has only four basic operations:

- Insert pairs into the collection.
- Delete pairs from the collection.
- Update the values of existing pairs.
- Find a value associated with a particular key. If there is no such value, then return an exception.

6.1 Schema Versus no Schema

SQL started off as a project at IBM under the name SEQUEL, which stood for Structured English-like Query Language; the structure came from the DDL. This is the sublanguage in SQL that defines the schema. Files are not anything like tables; rows are not records; columns are not fields. Rows are made up of columns. Columns have a known data type with constraints and are scalar. There is no ordering; you find a column by its name.

Likewise, tables are made up of rows. The rows have no ordering within a table; you find them with keys and not by a physical position. There are table-level constraints and relationships. Finally, the unit of work is the whole schema, which has intertable constraints and relationships (`CREATE ASSERTION`, and `FOREIGN KEY` and `PRIMARY KEY`).

An empty table still has structure! Its columns are still strongly typed and its constraints are still enforced. It just works out that all table-level predicates are true for an empty set. An empty file is, well, nothing. A blank reel of magnetic tape is an empty file. But so is a disk directory entry that has a name and a NIL pointer or other end-of-file marker as its only data. All empty files look and behave the same way; you read them and immediately get an end-of-file flag raised.

Having no schema puts all of the data integrity (if any!) in the application. Likewise, the presentation layer has no way to know what will come back to it. These systems are optimized for retrieval and appending operations, and often offer little functionality beyond record storage. The safety and query power of SQL systems are replaced by better scalability and performance for certain data models.

6.2 Query Versus Retrieval

NoSQL works with a huge quantity of data that does not need relational queries and integrity. The data can be structured, but that is a bonus, not a requirement. The classic example is an online store with a large inventory, and we want to pull up parts of a web page with graphics and a short text description in HTML when a customer gets to the website. Imagine an Internet shoe store as the example for the rest of this section.

This organization is particularly useful for statistical or real-time analysis of growing lists of elements, such as Twitter posts or the Internet server logs from a large group of users. There is no indexing or constraint checking; you deal with a huge pile of raw data. But it might not be clean.

6.3 Handling Keys

In this environment, how the keys are handled is vital. They have to be designed for the target data and not just a sequential numbering. For our shoe store, we will have some internal product identifier, but it will probably be so obscure that no customer is going to know it when he or she comes to the website. But if we can construct an encoding that the user can understand, then life is easier.

Good key encodings are not always easy to find. As Jorge Luis Borges comparatively stated in his essay “The Analytical Language of John Wilkins”:

> *These ambiguities, redundancies, and deficiencies recall those attributed by Dr. Franz Kuhn to a certain Chinese encyclopedia entitled Celestial Emporium of Benevolent Knowledge. On those remote pages it is written that animals are divided into (a) those that belong to the Emperor, (b) embalmed ones, (c) those that are trained, (d) suckling pigs, (e) mermaids, (f) fabulous ones, (g) stray dogs, (h) those that are included in this classification, (i) those that tremble as if they were mad, (j) innumerable ones, (k) those drawn with a very fine camel’s hair brush, (l) others, (m) those that have just broken a flower vase, (n) those that resemble flies from a distance.*

A dropdown list with these categories would not be much help for finding shoes on our website. There is no obvious key. Many years ago, I worked with a shoe company that was doing a data warehouse. The manufacturing side wanted reports based on the physical properties of the shoes and the marketing side wanted marketing categories. Steel-toed work boots were one category for manufacturing. However, at that time, marketing was selling them to two distinct groups: construction workers with big feet and Goth girls with small feet. These two groups did not shop at the same stores.

The ideal situation is a key that can quickly find which physical-commodity hardware device has the desired data. This is seldom possible in the real world unless your database deals with one subject area that has a classification system in place.

6.3.1 Berkeley DB

Berkeley DB (BDB), a software library, is the most widely used (< key >, < value >) database toolkit. Part of its popularity is that it can be used by so much other software. BDB is written in C with API bindings for C++, C#, PHP, Java, Perl, Python, Ruby, Tcl, Smalltalk, and most other programming

languages. The (< key >, < value >) pairs can be up to 4 gigabyte each and the key can have multiple data items.

This product began at the University of California at Berkeley in the late 1980s. It changed hands until Oracle bought out BDB in February 2006. The result is a bit of a legal fragmentation with it. While it is not relational, it has options for ACID transactions, with a locking system enabling concurrent access to data. There is a logging system for transactions and recovery. Oracle added support for SQL by including a version of SQLite. There is third-party support for PL/SQL in BDB via a commercial product named Metatranz StepSqlite.

A program accessing the database is free to decide how the data is to be stored in a record. BDB puts no constraints on the record's data. We are back to the file model where the host program gives meaning to the data.

6.3.2 Access by Tree Indexing or Hashing

There is nothing in the (< key >, < value >) model that prohibits using the usual tree indexes or hashing for access. The main difference is that targets might be on other hardware drives in a large "platter farm" of commodity hardware.

6.4 Handling Values

SQL is a strongly typed language of which columns have a known data type, default, and constraints. I tell people that 85–90% of the real work in SQL is done in the DDL. Bad DDL will force the programmers to kludge corrections in the DML over and over. The queries will repeat constraints in predicates that could have been written once in the DDL and picked up by the optimizer from the schema information tables.

The (< key >, < value >) model does not say anything about the nature of the values. The data can be structured, but when it is, the data types are usually simple, exact, and approximate numerics, strings, and perhaps temporal. There are no defaults or constraints. The purpose of this model is the ability to store and retrieve great quantities of data, *not* the relationships among the elements. These are records in the old FORTRAN file system sense of data, not rows from RDBMS.

The advantage is that any host language program can use the data immediately. If you look at the SQL standards, each ANSI X3J language has rules for converting its data types to SQL data types. There are also indicators

for handling nulls and passing other information between the host and the database.

6.4.1 Arbitrary Byte Arrays

Unstructured data is also kept in these products. The usual format is some kind of byte array: a BLOB, CLOB, or other large contiguous “chunks” of raw data. The host program decides what it means and how to use it. The (< key >, < value >) store simply wants to move it as fast as possible into the host.

The fastest way to move data is to have it in primary storage rather than on secondary storage. Notice I am using “primary” rather than “core,” “RAM,” etc., and “secondary” rather than “disk,” “drum,” “RAID,” etc. The technology is changing too fast and the line between primary (fast, direct addressing by processor) and secondary (slower, retrieved by other hardware) storage is getting blurred. My bet is that in a few years of 2015, the use of SSD (solid-state disk) will replace the moving disk. This will make radical changes in how we think of computing:

- There will be no difference between primary and secondary storage speeds.
- Cores (formerly known as CPUs) will also be so cheap, we will put them at all levels of the hardware. We will have mastered parallel programming.
- The results will be that a parallel table scan will be faster than using an index and not that different from hashing.

In short, everything becomes an in-memory database.

As of April 2013, IBM was sending out press releases that the hard drive would soon be dead in the enterprise. They put U.S. $1 billion in research to back up this prediction. IBM has launched a line of flash-based storage systems, called FlashSystems, based on technologies IBM acquired when it purchased Texas Memory in 2012. A set of FlashSystems could be configured into a single rack capable of storing as much as 1 petabyte of data, and capable of producing 22 million IOPS (input/output operations per second). Getting that same level of storage and throughput from a hard drive system would require 315 racks of high-performance disks.

IBM also has flash and disk hybrid storage systems, including the IBM Storwize V7000, IBM System Storage DS8870, and IBM XIV Storage

System. They realize that not all systems would benefit from the use of solid-state technologies right now. Performance has to be a critical factor for systems operations. As of 2013, generic hard drives cost about $2 per gigabyte, an enterprise hard drive costs about $4 per gigabyte, and a high-performance hard drive costs about $6 per gigabyte. A solid-state disk is $10 per gigabyte.

6.4.2 Small Files of Known Structure

Since (< key >, < value >) stores are used for websites, one of the values is a small file that holds a catalog page or product offering. XML and HTML are ideal for this. Every browser can use them—hypertext links are easy and they are simple text that is easy to update.

6.5 Products

This is a quick list of products as of 2013, taken from Wikipedia, using their classifications.

Eventually consistent (< key >, < value >) stores:

- Apache Cassandra
- Dynamo
- Hibari
- OpenLink Virtuoso
- Project Voldemort
- Riak

Hierarchical (< key >, < value >) stores:

- GT.M
- InterSystems Caché

Cloud or hosted services:

- Freebase
- OpenLink Virtuoso
- Datastore on Google Appengine
- Amazon DynamoDB

◆ Cloudant Data Layer (CouchDB)

(< key >, < value >) cache in RAM:

◆ Memcached

◆ OpenLink Virtuoso

◆ Oracle Coherence

◆ Redis

◆ Nanolat Database

◆ Hazelcast

◆ Tuple space

◆ Velocity

◆ IBM WebSphere eXtreme Scale

◆ JBoss Infinispan

Solid-state or rotating-disk (< key >, < value >) stores:

◆ Aerospike

◆ BigTable

◆ CDB

◆ Couchbase Server

◆ Keyspace

◆ LevelDB

◆ MemcacheDB (using BDB)

◆ MongoDB

◆ OpenLink Virtuoso

◆ Tarantool

◆ Tokyo Cabinet

◆ Tuple space

◆ Oracle NoSQL Database

Ordered (< key >, < value >) stores:

◆ Berkeley DB

- IBM Informix C-ISAM
- InfinityDB
- MemcacheDB
- NDBM

Concluding Thoughts

Because this is one of the simplest and most general models for data retrieval, there are lots of products that use it. The real trick in this technique is the design of the keys.

CHAPTER 7

Textbases

Abstract

Textbase is the current buzzword for document management systems, which deals with data kept in text as opposed to traditional structured data, relationships, or temporal models. It is the oldest form of data we use. Documents can be free text or semi-structured documents. The problem is that text can be treated as strings that have only syntax; that is patterns of characters that can be mathematically defined and mechanically manipulated by relatively simple algorithms. However, words have semantics; this requires human judgment or insanely complicated algorithms that are able to learn and make humanlike judgments. Most of the important business rules (laws, contracts, rules, definitions, communications, etc.) are in text.

Keywords

CQL (Contextual Query Language); KWOC; microfilm; NISO (National Information Standards Organization); regular expressions; Unicode

Introduction

I coined the term *textbase* decades ago to define an evolution that was just beginning. The most important business data is not in databases or files; it is in text. It is in contracts, warranties, correspondence, manuals, and reference material. Traditionally, data processing (another old term) dealt with highly structured data in a machine-usable media. Storage was expensive and you did not waste it with text. Text by its nature is fuzzy and bulky; traditional data is encoded to be precise and compact.

Text and printed documents also have legal problems. Thanks to having been invented several thousand years ago, they have requirements and traditions that are enforced by law and social mechanisms. As storage got cheaper and we got smarter, the printed word started to get automated. Not just the printed word, but reading and understanding it is also becoming automated.

7.1 Classic Document Management Systems

Textbases began as document management systems. The early ancestors were microfilm and microfiche. The text was stored as a physical image with a machine searchable index. In 1938, University Microfilms International (UMI) was established by Eugene Power to distribute microfilm editions of current and past publications and academic dissertations. They dominated this field into the 1970s.

The big changes were legal rather than technological. First, FAX copies of a document became legal, then microfilm copies of signed documents became legal, and, finally, electronically signed copies of signed documents became legal. The legal definition of an electronic signature can vary from a simple checkbox, a graphic file image of a signature, to an encryption protocol that can be verified by a third party. But the final result has been you did not need to have warehouses full of paper to have a legal document.

7.1.1 Document Indexing and Storage

Roll microfilm had physical “blips” between frames so that a microfilm reader could count and locate frames. Microfiche uses a Hollerith card (yes, they actually say “Hollerith” in the literature!) with a window of photographic film in it. Punch card sorters and storage cabinets can handle them. These are the original magnetic tape and punch card logical models! This should not be a surprise; new technology mimics the old technology. You do not jump to a new mindset all at once.

Later, we got hardware that could *physically* move the microfilm or microfiche around for us. They whiz, spin, and hum with lots of mechanical parts. If you can find an old video of these machines, you will get the feeling of what “steam punk” would be like if this science fiction genre were set in the 1950s instead of the Victorian era. We would call it “electromechanical punk” and everyone would wear gray flannel suits with narrow neckties.

These are a version of the (< key >, < value >) model used by NoSQL, using a more physical technology. But there is a difference; in textbases, the final judgment is made by a human reading and understanding the document. For semi-structured documents, such as insurance policies, there are policy numbers and other traditional structured data elements. But there are also semi-structured data elements, such as medical exams. And there are completely free text elements, such as a note like “Investigate this guy for fraud! We think he killed his wife for the insurance!” or worse.

7.1.2 Keyword and Keyword in Context

How do you help a human understand the document? The first contact a human has with a document is the title. This sounds obvious and it is why journals and formal documents had dull but descriptive titles. Before the early 20th century, books also had a secondary title to help someone decide to buy the book. Juvenile fiction was particularly subject to this (e.g., *The Boy Aviators in Nicaragua or In League with Insurgents*), but so was adult fiction (e.g., *Moby-Dick or The Whale*) and scholarly books (e.g., *An Investigation of the Laws of Thought on Which Are Founded the Mathematical Theories of Logic and Probabilities*).

The next step in the evolutionary process was a list of keywords taken from a specialized vocabulary. This is popular in technical journals. The obvious problem is picking that list of words. In a technical field, just keeping up with the terminology is hard. Alternate spellings, acronyms, and replacement terms occur all the time. Talk to a medical records person about the names of diseases over time.

Since the real data is in the semantics, the next step in evolution is the technique of keyword in context (KWIC) index. It was the most common format for concordance lines in documents. A concordance is an alphabetical list of the principal words used in a document with their immediate contexts. Before computers, they were difficult, time consuming, and expense. This is why they were only done for religious texts or major works of literature. The first concordance, to the *Vulgate Bible*, was compiled by Hugh of St. Cher (d. 1262), who employed 500 monks to assist him. In 1448, Rabbi Mordecai Nathan completed a concordance to the *Hebrew Bible*. It took him ten years. Today, we use a text search program with much better response times.

The KWIC system and its relatives are based on a concept called *keyword in titles*, which was first proposed for Manchester libraries in 1864 by Andrea Crestadoro. KWIC indexes divide the lines vertically in two columns with a strong gutter, with the keywords on the right side of the gutter in alphabetical order. For the nontypesetters, a gutter is a vertical whitespace in a body of text.

There are some variations in KWIC, such as KWOC (keyword out of context), KWAC (keyword augmented in context), and KEYTALPHA (key term alphabetical). The differences are in the display to the user. The good side of the keyword indexing methods is that they are fast to create once you have a keyword list. It is just as easy as indexing a traditional file or database. Many systems do not even require controlled vocabulary; they do a controlled

scan and build the list.

The bad news (and another step in the evolution) is that there no semantics in the search. The concept of related subjects, either narrower or broader concepts and topics in some recognized hierarchical structure, does not exist outside the mind of the users.

7.1.3 Industry Standards

Serious document handling began with librarians, not with computer people. This is no surprise; books existed long before databases. The National Information Standards Organization (NISO) and now the ANSI Z39 group, was founded in 1939, long before we had computers. It is the umbrella for over 70 organizations in the fields of publishing, libraries, IT, and media organizations. They have a lot of standards for many things, but the important one for us deals with documents, not library bookshelves.

Contextual Query Language

NISO has defined a minimal text language, the Common Query Language (CQL; (*http://zing.z3950.org/cql/intro.html*). It assumes that there is a set of documents with a computerized interface that can be queries. Queries can be formed with the usual three Boolean operators—`AND`, `OR`, and `NOT`—to find search words or phrases (strings are enclosed in double quotemarks) in a document. For example:

```
dinosaur NOT reptile

(bird OR dinosaur) AND (feathers OR scales)

“feathered dinosaur” AND (yixian OR jehol)
```

All the Boolean operators have the same precedence, and associate from left to right. This is not what a programmer expects! This means you need to use a lot of extra parentheses. These are the same queries:

```
dinosaur AND bird OR dinobird

(dinosaur AND bird) OR dinobird
```

Proximity operators select candidates based on the positional relationship among words in the documents. Here is the BNF (Backus Normal Form or Backus-Naur Form, a formal grammar for defining programming languages) for the infixed operator:

```
PROX/[< relation >] / [< distance >] / [< unit >] /
  [< ordering >]
```

However, any or all of the parameters may be omitted if the default values are required. Further, any trailing part of the operator consisting entirely of slashes (because the defaults are used) is omitted. This is easier to explain with the following examples.

```
foo PROX bar
```

Here, the words `foo` and `bar` are immediately adjacent to each other, in either order.

```
foo PROX///SENTENCE bar
```

The words `foo` and `bar` occur anywhere in the same sentence. This means that the document engine can detect sentences; this is on the edge of syntax and semantics. The default distance is zero when the unit is not a word.

```
foo PROX//3/ELEMENT bar
```

The words `foo` and `bar` must occur within three elements of each other; for example, if a record contains a list of authors, and author number 4 contains `foo` and author number 7 contains `bar`, then this search will find that record.

```
foo PROX/=/2/PARAGRAPH bar
```

Here, the words `foo` and `bar` must appear exactly two paragraphs apart—it is not good enough for them to appear in the same paragraph or in adjacent paragraphs. And we now have a paragraph as a search unit in the data.

```
foo PROX/>/4/WORD/ORDERED bar
```

This finds records in which the words appear, with `foo` first, followed more than four words later by `bar`, in that order. The other searches are not ordered.

A document can have index fields (year, author, ISBN, title, subject, etc.) that can also be searched. Again examples will give you an overview:

```
YEAR > 1998
TITLE ALL “complete dinosaur”
TITLE ANY “dinosaur bird reptile”
TITLE EXACT “the complete dinosaur”
```

The `ALL` option looks for all of the words in the string, without regard to their order. The `ANY` option looks for any of the words in the string. The `EXACT` option looks for all of the words in the string and they have to be in the order given. The exact searches are most useful on structured fields, such as ISBN codes, telephone numbers, and so forth. But we are getting into semantics with modifiers. The usual backslash modifier notation is used:

- `STEM`: The words in the search term are scanned and derived words from the same stem or root word are matched. For example, “walked,” “walking,” “walker,” and so forth would be reduced to the stem word “walk” in the search. Obviously this is language-dependent.
- `RELEVANT`: The words in the search term are in an implementation-dependent sense relevant to those in the records being searched. For example, the search subject `ANY/RELEVANT “fish frog”` would find records of which the subject field included any of the words “shark,” “tuna,” “coelocanth,” “toad,” “amphibian,” and so forth.
- `FUZZY`: A catch-all modifier indicating that the server can apply an implementation-dependent form of “fuzzy matching” between the specified search term and its records. This may be useful for badly spelled search terms.
- `PHONETIC`: This tries to match the term not only against words that are spelled the same but also those that sound the same with implementation-dependent rules. For example, with `SUBJECT =/PHONETIC`, “rose” might match “rows” and “roes” (fish eggs).

The modifier `EXACT/FUZZY` appears strange, but is very useful for structured fields with errors. For example, for a telephone number that might have incorrect digits, the structure is the exact part; the digits are the fuzzy part:

```
telephone_nbr EXACT/FUZZY “404-845-7777”
```

Commercial Services and Products

There have been many commercial services that provide access to data. The most used ones are LexisNexis and WestLaw legal research services. They have terabytes of online data sold by subscriptions. There are also other industry-specific database services, university-based document textbases, etc.

These services have a proprietary search language, but most of these languages are similar to the CQL with minor differences in syntax. They have the Boolean and proximity constructs, but often include support for their particular specialization, such as a legal or medical vocabulary.

Later there are products that can create textbases from raw data for a user. ZyIndex has been the leader in this area for decades. It was written in 1983 in Pascal as a full-text search program for files on IBM-compatible PCs running DOS. Over the years the company added optical character recognition (OCR), full-text search email and attachments, XML, and other features. Competition

followed and the internals vary, but most of the search languages stuck to the CQL model.

These products have two phases. The first phase is to index the documents so they can be searched. Indexing can also mean that the original text can be compressed; this is important! The second phase is the search engine. Indexing either ignores “noise words” or uses the full text. The concept of noise words is linguistic; there are “full words” and “empty words” in Chinese grammar. The noise or empty words are structural things like conjunctions, prepositions, articles, etc., as opposed to verbs and nouns that form the skeleton of a concept. Virtually, every sentence will have a noise word in it, so looking for them is a waste. The trade-off is that “to be or not to be” or other phrases are all noise words and might not be found.

There are systems that will scan every word when they index. The usual model is to associate a list of data page numbers instead of a precise location within the file. Grabbing the document in data pages is easy because that it how disk storage works. It is cheap and simple to decompress and display the text once it is in the main storage.

Regular Expressions

Regular expressions came from UNIX by way of the mathematician Stephen Cole Kleene. They are abstract pattern strings that describe a set of strings. ANSI/ISO standard SQL has a simple `LIKE` predicate and more complex SIMILAR TO predicate.

A vertical bar separates alternatives: `gray|grey` can match “gray” or “grey” as a string. Parentheses are used to define the scope and precedence of the operators (among other uses). A quantifier after a token (e.g., a character) or group specifies how often that preceding element is allowed to occur. The most common quantifiers are the question mark (`?`), asterisk (`*`) (derived from the Kleene star), and the plus sign (`+`) (Kleene cross). In English, they mean:

?	Zero or one of the preceding element: `colou?r` matches both “color” and “colour.”
*	Zero or more of the preceding element: `ab*c` matches “ac”, “abc”, “abbc”, “abbbc”, etc.
+	One or more of the preceding element: `ab + c` matches “abc”, “abbc”, “abbbc”, etc.

These constructions can be combined to form arbitrarily complex expressions. The actual syntax for regular expressions varies among tools. Regular expressions are pure syntax and have no semantics. The SQL people love this, but textbase people think in semantics, not syntax. This is why textbases do not use full regular expressions—a fundamental difference.

The IEEE POSIX Basic Regular Expressions (BRE) standard (ISO/IEC 9945-2:1993) was designed mostly for backward compatibility with the traditional (Simple Regular Expression) syntax but provided a common standard that has since been adopted as the default syntax of many UNIX regular expression tools, though there is often some variation or additional features. In the BRE syntax, most characters are treated as literals—they match only themselves (e.g., "a" matches "a"). The exceptions, listed in Table 7.1, are called metacharacters or metasequences.

Table 7.1

Exceptions to BRE (Basic Regular Expressions) Exceptions

Metacharacter	Description
`.`	Matches any single character (many applications exclude new lines, and exactly which characters are considered new lines is flavor-, character encoding–, and platform-specific, but it is safe to assume that the line-feed character is included). Within POSIX bracket expressions, the dot character matches a literal dot. For example, `a.c` matches "abc", but `[a.c]` matches only "a", ".", or "c".
`[ ]`	Matches a single character that is contained within the brackets. For example, `[abc]` matches "a", "b", or "c". `[a-z]` specifies a range that matches any lowercase letter from "a" to "z". These forms can be mixed: `[abcx-z]` matches "a", "b", "c", "x", "y", or "z", as does `[a-cx-z]`.
`[^ ]`	Matches a single character that is not contained within the brackets. For example, `[^abc]` matches any character other than "a", "b", or "c". `[^a-z]` matches any single character that is not a lowercase letter from "a" to "z". Likewise, literal characters and ranges can be mixed.
`^`	Matches the starting position within the string. In line-based tools, it matches the starting position of any line.
`$`	Matches the ending position of the string or the position just before a string-ending new line. In line-based tools, it matches the ending position of any line.
`\n`	Matches what the *n*th marked subexpression matched, where *n* is a digit from 1 to 9. This construct is theoretically irregular and was not adopted in the POSIX ERE syntax. Some tools allow referencing more than nine capturing groups.
`*`	Matches the preceding element zero or more times. For example, `ab*c` matches "ac", "abc", "abbbc", etc.; `[xyz]*` matches "," "x", "y", "z", "zx", "zyx", "xyzzy", and so on; `(ab)*` matches "," "ab", "abab", "ababab", and so on.

7.2 Text Mining and Understanding

So far, what we have discussed are simple tools that use a local textbase to do retrievals. While retrieval of text is important, this is not what we really want to do with text. We want meaning, not strings. This is why the first tools were used by lawyers and other professionals who have special knowledge, use a specialized vocabulary, and draw text from limited sources.

If you want to play with a large textbase, Google has an online tool

(*http://books.google.com/ngrams*) that will present a simple graph of the occurrences of a search phrase in books from 1800 to 2009 in many languages. This lets researchers look for the spread of a term, when it fell in and out of fashion and so forth.

We had to get out of this limited scope and search everything—this led to Google, Yahoo, and other web search engines. The volume of text and the constant flow of it are the obvious problems. The more subtle problems are in the mixed nature of text, but even simple searches on large volumes of text are time consuming.

Google found a strong correlation between how many people search for flu-related topics and how many people actually have flu symptoms. Of course, not every person who searches for "flu" is actually sick, but a pattern emerges when all the flu-related search queries are added together. Some of the details are given in *http://www.nature.com/nature/journal/v457/n7232/full/nature07634.html*.

7.2.1 Semantics versus Syntax

Strings are simply patterns. You do not care if the alphabet used to build strings is Latin, Greek, or symbols that you invented. The most important property of a string is linear ordering of the letters. Machines love linear ordering and they are good at parsing when given a set of syntax rules. We have a firm mathematical basis and lots of software for parsing. Life is good for strings.

Words are different—they have meaning. They form sentences; sentences form paragraphs; paragraphs form a document. This is semantics. This is reading for understanding. This was one of the goals for artificial intelligence (AI) in computer science. The computer geek joke has been that AI is everything in computer research that *almost* works, but not quite.

Programs that read and grade student essays are called *robo-readers* and they are controversial. On March 15, 2013, Les Perelman, the former director of writing at the Massachusetts Institute of Technology, presented a paper at the Conference on College Composition and Communication in Las Vegas, NV. It was about his critique of the 2012 paper by Mark D. Shermis, the dean of the college of education at the University of Akron. Shermis and co-author Ben Hamner, a data scientist, found automated essay scoring was capable of producing scores similar to human scores when grading SAT essays.

Perelman argued the methodology and the data in the Shermis–Hamner

paper do not support their conclusions. The reason machine grading is important is that the vast majority of states are planning to introduce new high-stakes tests for K–12 students that require writing sections. Educational software is marketed now because there simply are not enough humans to do the work.

Essay grading software is produced by a number of major vendors, including McGraw-Hill and Pearson. There are two consortia of states preparing to introduce totally new high-stakes standardized exams in 2014 to match the common core curriculum, which has swept the nation. The two consortia—the Partnership for Assessment of Readiness for College and Careers, and Smarter Balanced Assessment Consortium—want to use machines to drive down costs.

Perelman thinks teachers will soon teach students to write to please robo-readers, which Perelman argues disproportionately gives students credit for length and loquacious wording, even if they do not quite make sense. He notes, “The machine is rigged to try to get as close to the human scores as possible, but machines don’t understand meaning.” In 2010, when the common core exams were developed for its 24 member states, the group wanted to use machines to grade all of the writing. Today, this has changed. Now, 40% of the writing section, 40% of the written responses in the reading section, and 25% of the written responses in the math section will be scored by humans.

Many years ago, *MAD* magazine did a humor piece on high school teachers grading famous speeches and literature. Abraham Lincoln was told to replace “four score and seven years” with “87 years” for clarity; Ernest Hemingway needed to add more descriptions; Shakespeare was told that “to be or not to be” was contradictory and we needed a clear statement of this question. “The technology hasn’t moved ahead as fast as we thought,” said Jacqueline King, a director at Smarter Balanced. I am not sure that it can really move that far ahead.

7.2.2 Semantic Networks

A semantic network or semantic web is a graph model (see Chapter 3 on graph databases) of a language. Roughly speaking, the grammar is shown in the arcs of the graph and the words are the nodes. The nodes need a disambiguation process because, to quote the lyrics from “Stairway to Heaven” (1971) by Led Zeppelin, “cause you know sometimes words have two meanings,” and the software has to learn how to pick a meaning.

Word-sense disambiguation (WSD) is an open problem of natural language processing that has two variants: *lexical sample* and *all words* task. The lexical sample uses a small sample of preselected words. The all words method uses all regular expressions that came from UNIX by way of the mathematician Stephen Cole Kleene. They are abstract pattern strings that describe a set of strings. ANSI/ISO standard SQL has a simple `LIKE` predicate and more complex `SIMILAR TO` predicate. This is a more realistic form of evaluation, but it is more expensive to produce.

Consider the (written) word “bass”:

- A type of fish
- Low-frequency musical tones

If you heard the word spoken, there would be a difference in pronunciation, but these sentences give no such clue:

- I went *bass* fishing.
- The *bass* line of the song is too weak.

You need a context. If the first sentence appears in text, a parse can see that this might be a adjective–noun pattern. But if it appears in an article in *Field & Stream* magazine, you have more confidence in the semantics of “bass” meaning a fish than if it were in *Downbeat* music reviews.

WordNet is a lexical database for the English language that puts English words into sets of synonyms called *synsets*. There are also short, general definitions and various semantic relations between these synonym sets. For example, the concept of “car” is encoded as {“car”, “auto”, “automobile”, “machine”, “motorcar”}. This is an open-source database that is widely used.

7.3 Language Problem

Americans tend not to learn foreign languages. Culturally, we believe that everyone learns American English and this is largely true today. We inherited the old British Empire’s legacy and added U.S. technical and economic domination to it. Our technical domination gave us another advantage when computers came along. We had the basic Latin alphabet embedded in ASCII so it was easy to store English text on computers. Alphabets are linear and have a small character set; other languages are not so lucky. Chinese requires a huge character set; Korean and other languages require characters that are written in two dimensions.

7.3.1 Unicode and ISO Standards

The world is made up of more alphabet, syllabary, and symbol systems than just Latin-1. We have Unicode today. This is a 16-bit code that can represent in most of the world's writing systems. It has a repertoire of more than 110,000 characters covering 100 scripts. The standard shows the printed version of each character, rules for normalization (how to assemble a character from accent marks and other overlays), decomposition, collation, and display order (right-to-left scripts versus left-to-right scripts).

Unicode is the de-facto and de-jure tool for internationalization and localization of computer software. It is part of XML, Java, and the Microsoft .NET framework. Before Unicode, the ISO 8859 standard defined 8-bit codes for most of the European languages, with either complete or partial coverage. The problem was that a textbase with arbitrary scripts mixed with each other could not be stored without serious programming. The first 256 code points are identical to ISO 8859-1. This lets us move existing western text to Unicode, but also to store ISO standard encodings, which are limited to this subset of characters.

Unicode provides a unique numeric code point for each character. It does not deal with display, so SQL programmers will like that abstraction. The presentation layer has to decide on size, shape, font, colors, or any other visual properties.

7.3.2 Machine Translation

Languages can have inflectional and agglutinative parts in their grammar. An inflectional language changes the forms of their words to show grammatical functions. In linguistics, declension is the inflection of nouns, pronouns, adjectives, and articles to indicate number (at least singular and plural, but Arabic also has dual), case (nominative, subjective, genitive, possessive, instrumental, locational, etc.), and gender. Old English was a highly inflected language, but its declensions greatly simplified as it evolved into modern English.

An example of a Latin noun declension is given in Table 7.2, using the singular forms of the word *homo*.

Table 7.2

Latin Declension for *Homo*

homo	(nominative) subject
hominis	(genitive) possession
hominī	(dative) indirect object
hominem	(accusative) direct object
homine	(ablative) various uses
—	(vocative) addressing a person
—	(locative) rare for places only

There are more declensions and Latin is not the worst such language. These languages are hard to parse and tend to be highly irregular. They often have totally different words for related concepts, such as plurals. English is notorious for its plurals (mouse versus mice, hero versus heroes, cherry versus cherries, etc.).

At the other extreme, an agglutinative language forms words by pre- or post- fixing morphemes to a root or stem. In contrast to Latin, the most agglutinate language is Esperanto. The advantage is that such languages are much easier to parse. For example:

"I see two female kittens"—Mi vidas du katinidojn

where kat/in/id/o/j/n means "female kittens in accusative" and made of:

kat (cat = root word) || in (female gender affix) || id ("a child" affix) || o (noun affix) || j (plural affix) || n (accusative case affix)

The root and the affixes never change. This is why projects like Distributed Language Translation (DLT) use a version of Esperanto as an intermediate language. The textbase is translated into the intermediate language and then from intermediate language into one of many target languages. Esperanto is so simple that intermediate language can be translated at a target computer. DLT was used for scrolling video news text on cable television in Europe.

Concluding Thoughts

We are doing a lot of things with text that we never thought would be possible. But the machines cannot make intuitive jumps; they are narrow in specialized worlds.

In 2011, IBM entered their Watson AI computer system on the television quiz show *Jeopardy*. The program had to answer questions posed in natural language. It was very impressive and won a million dollars in prize money. The machine was specifically developed to answer questions on *Jeopardy*; it was not a general-purpose tool. Question categories that used clues containing

only a few words were problematic for it. In February 2013, Watson's first commercial application is at the Memorial Sloan–Kettering Cancer Center in conjunction with health insurance company WellPoint. It is now tuned to do utilization management decisions in lung cancer treatments.

With improved algorithms and computing power, we are getting to the point of reading and understanding the text. That is a whole different game, but it is not human intelligence yet. These algorithms *find existing relationships*, they do not *create* a new one. My favorite example from medicine was a burn surgeon who looked at the expanded metal grid on his barbeque and realized that the same pattern of slits and stretching could be used with human skin to repair burns. Machines do not do that.

LexisNexis and WestLaw teach law students how to use their textbases for legal research. The lessons include some standard exercises based on famous and important legal decisions. Year after year, the law students overwhelming fail to do a correct search. They fetch documents they do not need and fail to find documents they should have found. Even smart people working in a special niche are not good at asking questions from textbases.

References

1. Perelman LC. *Critique (Ver 3.4) of Mark D Shermis & Ben Hammer, "Contrasting state-of-the-art automated scoring of essays: Analysis"*. 2012; In: *http://www.scoreright.org/NCME_2012_Paper3_29_12.pdf*; 2012.
2. Ry Rivard Ty. *Humans fight over robo-readers*. 2013, Mar 13; In: *http://www.insidehighered.com/news/2013/03/15/professors-odds-machine-graded-essays#ixzz2cnutkboG*; 2013, Mar 13;.
3. Shermis MD, Hamner B. *Contrasting state-of-the-Art automated scoring of essays: Analysis The University of Akron*. 2012; In: *http://www.scoreright.org/NCME_2012_Paper3_29_12.pdf*; 2012.

CHAPTER 8

Geographical Data

Abstract

Geographical data, geospatial, or spatiotemporal databases deal with geography. Most of the queries deal with quantities, densities, and contents within a geographical area. While we learned longitude and latitude in school, there are other methods for locating positions on Earth. In particular, HTM is much more accurate and better suited for satellites. GISs also have to integrate traditional static data into GIS indexes, such as the names of businesses with their locations. But it also has to include dynamic and temporal information. In particular, queries that deal with flow and time, such as traffic patterns, are difficult.

Keywords

geographical information system (GIS); geographical data; Canada Geographic Information System (CGIS); ISO 19106 standards; proximity relationships; longitude; latitude; hierarchical triangular mesh (HTM)

Introduction

Geographic information systems (GISs) are databases for geographical, geospatial, or spatiotemporal (space–time) data. This is more than cartography. We are not just trying to locate something on a map; we are trying to find quantities, densities, and contents of things within an area, changes over time, and so forth.

This type of database goes back to early epidemiology with “Rapport sur La Marche Et Les Effets du Choléra Dans Paris et Le Département de la Seine,” prepared by French geographer Charles Picquet in 1832. He made a colored map of the districts of Paris to show the cholera deaths per 1,000 population.

In 1854, John Snow produced a similar map of the London cholera outbreak, using points to mark some individual cases. This study is a classic story in statistics and epidemiology. London in those days had public hand-operated water pumps; citizens hauled water back to their residences in buckets. His map clearly showed the source of the disease was the Broad

Street pump, which had been contaminated with raw sewage. Cholera begins with watery diarrhea with a fluid loss of as much as 20 liters of water a day, leading to rapid dehydration and death for 70% of victims. Within days, cholera deaths dropped from hundreds to almost none. We had cartography for centuries, but the John Snow map was one of the first to analyze clusters of geographically dependent phenomena.

Please remember that these maps were all produced by hand. We did not have photographic technology until the early 20th century and computer graphics until the late 20th century. The first true GIS was the Canada GIS (CGIS), developed by the Canadian Department of Forestry and Rural Development. Dr. Tomlinson, the head of this project, become known as the "father of GIS" not just for the term *GIS*, but for his use of overlays for spatial analysis of convergent geographic data. Before this, computer mapping was just maps without analysis. The bad news is that it never became a commercial product.

However, it inspired all of the commercial products that followed it. Today, the big players are Environmental Systems Research Institute (ESRI), Computer-Aided Resource Information System (CARIS), Mapping Display and Analysis System (MIDAS, now MapInfo), and Earth Resource Data Analysis System (ERDAS). There are also two public-domain systems (MOSS and GRASS GIS) that began in the late 1970s and early 1980s.

There are geospatial standards from ISO Technical Committee 211 (ISO/TC 211) and the Open Geospatial Consortium (OGC). The OGC is an international industry consortium of 384 companies, government agencies, universities, and individuals that produce publicly available geoprocessing specifications and grade products on how well they conform to the specifications with compliance testing. The main goal is information interchange rather than query languages, but they include APIs for host languages and a Geography Markup Language (GML). Take a look at their web page for current information: *http://www.opengeospatial.org/standards/as*.

In practice, most queries are done graphically from a screen and not with a linear programming language. Answers are also typically a mix of graphics and traditional data presentations. Table 8.1 outlines some basic ISO standards.

Table 8.1

ISO Standards

ISO 19106:2004: Profiles	Simple features from GML.
ISO 19107:2003: Spatial Schema	See also ISO 19125 and ISO 19115; basic concepts defined in this standard are implemented. This is the foundation for simple features in GML.
ISO 19108:2003: Temporal Schema	Time-aware data in GIS.
ISO 19109:2005: Rules for Application Schema	A conceptual schema language in UML.
ISO 19136:2007: Geography Markup Language	This is equivalent to OGC GML 3.2.1.

8.1 GIS Queries

So much for the history and tools. What makes a GIS query different from a basic SQL query? When we ask a typical SQL query, we want aggregated numeric summaries of data measured in scalar units. In GIS we want spatial answers and space is not linear or scalar.

8.1.1 Simple Location

The most primitive GIS query is simply “Where am I?” But you need more than “Here!” as an answer. Let’s imagine two models for locations. If we are in a ship or airplane, the answer has a time and direction to it. There is an estimate of arrival at various points in the journey. However, if we are in a lighthouse, we will stay in the same location for a very long time! Even cities move: Constantinople, Istanbul, and Byzantium (among other names!) are historical names for the same location. Oh, wait! Their locations also changed over time.

8.1.2 Simple Distance

Once you have a way to describe any location, then the obvious next step is the distance between two of them. But distance is not so simple. Americans have the expression “as the crow flies” for direct linear distance. But distance is not linear.

The actual distance can be airplane, road, or railroad travel distances. The distance can be expressed as travel hours. Distance can be effectively infinite. For example, two villages on opposite sides of a mine field can be unreachable.

8.1.3 Find Quantities, Densities, and Contents within an Area

The next class of queries deals with area. Or is it volume? In traditional cartography, elevation was the third dimension and we had topographic maps with contour lines. But in GIS queries, the third dimension is usually something else.

The television show *Bar Rescue* on SPIKE network features Jon Taffer remodeling failing bars. He is an expert in this field and the show gives a lot of the science of bar management. Almost every show has a section where Taffer spreads out a map of the neighborhood and gives a GIS analysis of the client's situation.

The first query is always measurements of the neighborhood. How many other bars are in the neighborhood (quantity)? Are they clustered or spread out (density)? How many of them are like your bar (content)? These are static measurements that can be found in census and legal databases. What Taffer adds is experience and analysis.

8.1.4 Proximity Relationships

Proximity relationships are more complex than the walking distance between bars in the neighborhood. The concept of proximity is based on access paths. In New York, NY, most people walk, take a taxi, or take the subway; sidewalks are the access paths. In Los Angeles, CA, almost nobody walks; streets are the access paths (if there is parking).

The most common mistake in GIS is drawing a circle on a map and assuming this is your neighborhood. Most GIS systems can fetch the census data from the households within the circle. The trick is to change the circle into a polygon. You extend or contact the perimeter of your neighborhood based on other information.

8.1.5 Temporal Relationships

The simple measurement is linear distance, but travel time might be more important. Consider gas stations on interstate highway exits; the physical distance can be large, but the travel time can be short.

A GIS query for the shortest travel time for an emergency vehicle is not linear, nor is it road distance. The expected travel time also depends on the

date and time of day of the trip.

One of the best uses of GIS is an animation of geography changes over time. It can show growth and decay patterns. From those patterns, we can make predictions.

8.2 Locating Places

The first geographical problem is simply locating a place on Earth and finding out what is there. While we take geographical positioning systems (GPSs) and accurate maps for granted today, this was not always the case. For most of human history, people depended on the sun, moon, and stars for that data. Geographic location also depended on physical features that could be described in text.

This is fine for relatively short distances, but not for longer voyages and greater precision. We needed a global system.

8.2.1 Longitude and Latitude

Longitude and latitude should be a familiar concept from elementary school. You assume the world is a sphere (it really is not) rotating on an axis and draw imaginary lines on it. The lines from the North Pole to the South Pole are longitude. The circles that start at the Equator are latitude. This system goes back to the second-century BCE and the astronomer Hipparchus of Nicea, who improved Babylonian calculations. The Babylonians used base 60 numbers, so we have a system of angular measure based on a circle with 360 degrees with subdivisions of 60 units (Figure 8.1). Hipparchus used the ratio between the length of days in a year and solar positions, which can be done with a sundial.

FIGURE 8.1 Longitude and Latitude Lines. Source: *http://commons.wikimedia.org/wiki/Image:Sphere_filled_blue.svg rights*, GNU Free Documentation license, Creative Commons Attribution ShareAlike license.

Longitude was harder and required accurate clocks for the best and simplest solution. You set a clock to Greenwich Mean Time, take the local solar time, and directly compute the longitude. But clocks were not accurate and did not travel well on ships. A functional clock was finally built in the 1770 s by John Harrison, a Yorkshire carpenter. The previous method was the lunar distance method, which required lookup tables, the sun, the moon, and ten stars to find longitude at sea, but it was accurate to within half a degree. Since these tables were based on the Royal Observatory, Greenwich Mean Time became the international standard.

Most GIS problems are local enough to pretend that the Earth is flat. This avoids spherical trigonometry and puts us back into plane geometry. Hooray! We can draw maps from survey data and we are more worried about polygons and lines that represent neighborhoods, political districts, roads, and other human-made geographical features.

Unfortunately, these things were not that well measured in the past. In Texas and other larger western states, the property lines were surveyed 200 or more years ago (much of it done by the Spanish/Mexican period) with the equipment available at that time. Many of the early surveys used physical features that have moved; rivers are the most common example, but survey markers have also been lost or moved. Texas is big, so a small error in an angle or a boundary marker becomes a narrow wedge shape of no-man’s land between property lines. These wedges and overlaps are very large in parts of west Texas. They also become overlapping property boundaries.

But do not think that having modern GPS will prevent errors. They also have errors, though much fewer and smaller than traditional survey tools produce. Two receivers at adjacent locations often experience similar errors, but they can often be corrected statistically. The difference in coordinates between the two receivers should be significantly more accurate than the absolute position of each receiver; this is called *differential positioning*. GPS can be blocked by physical objects and we are not talking about big objects. Mesh fences or other small objects can deflect the satellite signal. Multipath errors occur when the satellite signal takes two routes to the receiver. Then we have atmospheric disturbances that do to GPSs what they do to your television satellite dish reception.

These surveys need to be redone. We have the technology today to establish the borders and corners of the polygon that define a parcel of land far better than we did even ten years ago. But there are legal problems concerning who would get title to the thousands of acres in question and what

jurisdictions the amended property would belong to. If you want to get more information, look at the types of land surveys at *http://hovell.net/types.htm*.

8.2.2 Hierarchical Triangular Mesh

An alternative to (longitude, latitude) pairs is the hierarchical triangular mesh (HTM). While (longitude, latitude) pairs are based on establishing a point in a two-dimensional coordinate system on the surface of the Earth, HTM is based on dividing the surface into almost-equal-size triangles with a unique identifier to locate something by a containing polygon. The HTM index is superior to cartographical methods using coordinates with singularities at the poles.

If you have seen a geodesic dome or Buckminster Fuller's map, you have some feeling for this approach. HTM starts with an octahedron at level zero. To map the globe into an octahedron (regular polyhedron of eight triangles), align it so that the world is first cut into a northern and southern hemisphere. Now slice it along the prime meridian, and then at right angles to both those cuts. In each hemisphere, number the spherical triangles from 0 to 3, with either "N" or "S" in the prefix.

A triangle on a plane always has exactly 180°, but on the surface of a sphere and other positively curved surfaces, it is always greater than 180°, and less than 180° on negatively curved surfaces. If you want a quick mind tool, think that a positively curved surface has too much in the middle. A negatively curved surface is like a horse saddle or the bell of a trumpet; the middle of the surface is too small and curves the shape.

The eight spherical triangles are labeled N0 to N3 and S0 to S3 and are called "level 0 trixels" in the system. Each trixel can be split into four smaller trixels recursively. Put a point at the middle of each edge of the triangle. Use those three points to make an embedded triangle with great circle arc segments. This will divide the original triangle into four more spherical triangles at the next level down. Trixel division is recursive and smaller and smaller trixels to any level you desire (Figures 8.2–8.4).

FIGURE 8.2 Second Level Tessellation.

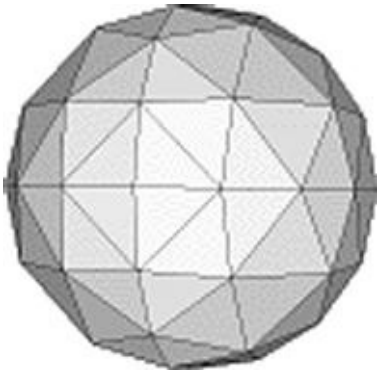


FIGURE 8.3 Third Level Tessellation.

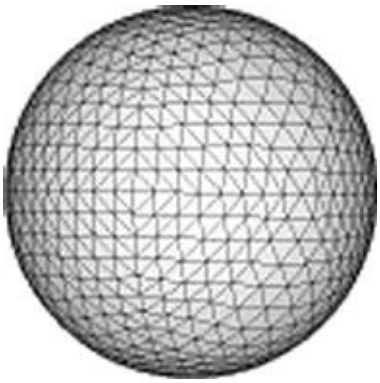


FIGURE 8.4 High Level Tessellation.

To name the new trixels, take the name of the patent trixel and append a digit from 0 to 3 to it, using a counterclockwise pattern. The (n = {0, 1, 2}) point on the corner of the next level is opposite the same number on the corner of the previous level. The center triangle always gets a 3.

The triangles are close to the same size at each level. As they get smaller, the difference also decreases. At level 7 approximately three-quarters of the trixels are slightly smaller than the average size, and one-quarter are larger. The difference is because the three corner trixels (0, 1, 2) are smaller than trixel 3 in the center one. Remember that geometry lesson about triangles on spheres at the start of this section? The ratio of the maximum over the

minimum areas at higher depths is about two. Obviously, smaller trixels have longer names and the length of the name gives its level.

The name can be used to compute the exact location of the three vertex vectors of the given trixel. They can be easily encoded into an integer identifier by assigning 2 bits to each level and encoding "N" as 11 and "S" as 10. For example, N01 encodes as binary 110001 or hexadecimal 0x31. The HTM ID (HtmID) is the number of a trixel (and its center point) as a unique 64-bit string (not all strings are valid trixels).

While the recursion can go to infinite, the smallest valid HtmID is 8 levels but it is easy to go to 31 levels, represented in 64 bits. Level 25 is good enough for most applications, since it is about 0.6 meters on the surface of the Earth or 0.02 arc-seconds. Level 26 is about 30 centimeters (less than one foot) on the Earth's surface. If you want to get any closer than that to an object, you are probably trying to put a bullet in it or perform surgery.

The tricky stuff is covering an irregular polygon with trixels. A trixel can be completely inside a region, completely outside a region, overlapping the region, or touching another trixel. Obviously, the smaller the trixels used to approximate the region, the closer the approximation of the area. If we assume that the location of something is in the center of a trixel, then we can do some math or build a lookup table to get the distance between locations. Beyond depth 7, the curvature of the Earth becomes irrelevant, so distances within a trixel are simple Cartesian calculations. You spread the triangle out flat and run a string between the departure and destination triangles. You can get more math than you probably want in Microsoft research paper MSR-TR-2005-123.

8.2.3 Street Addresses

The United States and many other counties are moving to what we call the nine-one-one system. It assigns a street address that has a known (longitude, latitude) that can be used by emergency services, hence the name. Before this, an address like "The Johnson Ranch" was used in rural areas.

A database known as the Master Street Address Guide (MSAG) describes address elements, including the exact spellings of street names and street number ranges. It ties into the U.S. Postal Service's (USPS) Coding Accuracy Support System (CASS). You can get the basic details of CASS at *http://pe.usps.com/text/pub28*.

The use of U.S.-style addresses is not universal. In Korea, a big debate was

started in 2012 when the country switched from the Japanese system to the U.S. system. In the Japanese system, a town or city is divided into neighborhoods. Within each neighborhood, streets have local names and the buildings are numbered in the order they were built. Numbers are not reused when a building is destroyed. There is no way to navigate in such a system unless you already know where something is. There is no way to get geographical data from these addresses.

The objection in Korea came from Buddhist groups. Many of the neighborhoods were named after Buddhist saints and they were afraid those names would disappear when physically contiguous streets were give a uniform name.

8.2.4 Postal Codes

I cannot review all of the postal addresses on Earth. Showing a cultural basis, I will look at the United States, Canada, and the United Kingdom. They are all English-speaking nations with a common heritage, so you might expect some commonality. Wrong again.

Postal codes can be broadly lumped into those based on geography and those based on the postal delivery system. The U.S. ZIP code system is highly geographical, the Canadian system is less so, and the U.K. system is based on the post offices in the 1800 s.

8.2.5 ZIP Codes

The zone improvement plan (ZIP) code is a postal code system used by the USPS. The term *ZIP code* was a registered servicemark (a type of trademark) held by the USPS, but the registration expired. Today the term has become almost a generic name for any postal code, and almost nobody remembers what it originally meant. The basic format is a string of five digits that appear after the state abbreviation code in a U.S. address. An extended ZIP + 4 code was introduced in 1983, which includes the five digits of the ZIP code, a hyphen, and four more digits that determine a more precise location than the ZIP code alone.

The code is based on geography, which means that locations with the same ZIP code are physically contiguous. The first digit is a contiguous group of states.

The first three digits identify the sectional center facility (SCF). An SCF

sorts and dispatches mail to all post offices with those first three digits in their ZIP codes. Most of the time, this aligns with state borders, but there are some exceptions for military bases and other locations that cross political units or that can be best served by a closer SCF.

The last two digits are an area within the SCF. But this does not have to match to a political boundary. They tend to identify a local post office or station that serves a given area.

8.2.6 Canadian Postal Codes

Canada (*www.canadapost.ca*) was one of the last western countries to get a nationwide postal code system (1971). A Canadian postal code is a string of six characters that forms part of a postal address in Canada. This is an example of an alphanumeric system that has less of a geographical basis than the ZIP codes used in the United States. The format is

```
  postal_code SIMILAR TO '[:UPPER:][:DIGIT:][:UPPER:] [:DIGIT:]
[:UPPER:][:DIGIT:]'
```

Notice the space separating the third and fourth characters. The first three characters are a forward sortation area (FSA), which is geographical—for example, A0A is in Newfoundland and Y1A in the Yukon.

But this is not a good validation. The letters D, F, I, O, Q, and U are not allowed because handwritten addresses might make them look like digits or other letters when they are scanned. The letters W and Z are not used as the first letter. The first letter of an FSA code denotes a particular postal district, which, outside of Quebec and Ontario, corresponds to an entire province or territory. Owing to Quebec's and Ontario's large populations, those two provinces have three and five postal districts, respectively, and each has at least one urban area so populous that it has a dedicated postal district ("H" for Laval and Montréal, and "M" for Toronto).

At the other extreme, Nunavut and the Northwest Territories (NWT) are so small they share a single postal district. The digit specifies if the FSA is rural (zero) or urban (nonzero). The second letter represents a specific rural region, entire medium-size city, or section of a major metropolitan area.

The last three characters denote a local delivery unit (LDU). An LDU denotes a specific single address or range of addresses, which can correspond to an entire small town, a significant part of a medium-size town, a single side of a city block in larger cities, a single large building or a portion of a very large one, a single large institution such as a university or a hospital, or a

business that receives large volumes of mail on a regular basis. LDUs ending in zero are postal facilities, from post offices and small retail postal outlets all the way to sortation centers.

In urban areas, LDUs may be specific postal carriers' routes. In rural areas where direct door-to-door delivery is not available, an LDU can describe a set of post office boxes or a rural route. LDU 9Z9 is used exclusively for Business Reply Mail. In rural FSAs, the first two characters are usually assigned in alphanumerical order by the name of each community.

LDU 9Z0 refers to large regional distribution center facilities, and is also used as a placeholder, appearing in some regional postmarks such as K0H 9Z0 on purely local mail within the Kingston, Ontario, area.

8.2.7 Postcodes in the United Kingdom

These codes were introduced by the Royal Mail over a 15-year period from 1959 to 1974 and are defined in part by British Standard BS-7666 rules. Strangely enough they are also the lowest level of aggregation in census enumeration. U.K. postcodes are variable-length alphanumeric, making them hard to computerize. The format does not work well for identifying the main sorting office and suboffice. They seem to have been based on what the Royal Post looked like several decades or centuries ago by abbreviating local names at various times in the 15-year period they began.

Postcodes have been supplemented by a newer system of five-digit codes called Mailsort, but only for bulk mailings of 4,000 or more letter-size items. Bulk mailers who use the Mailsort system get a discount, but bulk deliveries by postcode do not.

Postcode Formats

The format of U.K. postcode is generally given by the regular expression:

```
  (GIR     0AA|[A-PR-UWYZ]([0-9]{1,2}|([A-HK-Y][0-9]|[A-HK-Y][0-9]
([0-9]|  [ABEHMNPRV-Y]))|[0-9][A-HJKS-UW])  [0-9][ABD-HJLNP-UW-Z]
{2})
```

It is broken into two parts with a space between them. It is a hierarchical system, working from left to right—the first letter or pair of letters represents the area, the following digit or digits represents the district within that area, and so on. Each postcode generally represents a street, part of a street, or a single address. This feature makes the postcode useful to route planning software.

The part before the space is the outward code, which identifies the destination sorting office. The outward code can be split further into the area part (letters identifying one of 124 postal areas) and the district part (usually numbers); the letters in the inward code exclude C, I, K, M, O, and V to avoid confusing scanners. The letters of the outward code approximate an abbreviation of the location (London breaks this pattern). For example, “L” is Liverpool, “EH” is Edinburgh, “AB” is Aberdeen, and “BT” is Belfast and all of Northern Ireland.

The remaining part is the inward code, which is used to sort the mail into local delivery routes. The inward code is split into the sector part (one digit) and the unit part (two letters). Each postcode identifies the address to within 100 properties (with an average of 15 properties per postcode), although a large business may have a single code.

Greater London Postcodes

In London, postal area postcodes are based on the 1856 system of postal districts. They do not match the current boundaries of the London boroughs and can overlap into counties in the greater London area. The numbering system appears arbitrary on the map because it is historical rather than geographical.

The most central London areas needed more postcodes than were possible in an orderly pattern, so they have codes like EC1A 1AA to make up the shortage. Then some codes are constructed by the government for their use, without regard to keeping a pattern. For example, in Westminster:

- SW1A 0AA: House of Commons
- SW1A 0PW: House of Lords, Palace of Westminster
- SW1A 1AA: Buckingham Palace
- SW1A 2AA: 10 Downing Street, Prime Minister and First Lord of the Treasury
- SW1A 2AB: 11 Downing Street, Chancellor of the Exchequer
- SW1A 2HQ: HM Treasury headquarters
- W1A 1AA: Broadcasting House
- W1A 1AB: Selfridges
- N81 1ER: Electoral Reform Society has all of N81

There are also nongeographic postcodes, such as outward code BX, so that they can be retained if the recipient changes physical locations. Outward codes beginning XY are used internally for misaddressed mail and international outbound mail. This does not cover special postcodes for the old Girobank, Northern Ireland, Crown dependencies, British Forces Post Office (BFPO), and overseas territories.

In short, the system is so complex that you require software and data files from the Royal Post (postcode address file, or PAF, which has about 27 million U.K. commercial and residential addresses) and specialized software. The PAF is not given out free by the Royal Mail, but licensed for commercial use by software vendors and updated monthly. In the United Kingdom, most addresses can be constructed from just the postcode and a house number. But not in an obvious way, so GISs have to depend on lookup files to translate the address into geographical locations.

8.3 SQL Extensions for GIS

In 1991, the National Institute for Science and Technology (NIST) set up the GIS/SQL work group to look at GIS extensions. Their work is available as "Towards SQL Database Language Extensions for Geographic Information Systems" at *http://books.google.com*.

In 1997, the Open Geospatial Consortium (OGC) published "OpenGIS Simple Features Specifications for SQL," which proposes several conceptual ways for extending SQL to support spatial data. This specification is available from the OGC website at *http://www.opengis.org/docs/99-049.pdf*. For example, PostGIS is an open-source, freely available, and fairly OGC-compliant spatial database extender for the PostgreSQL Database Management System. SQL Server 2008 has its spatial support: there is Oracle Spatial, and the DB2 spatial extender. These extensions add spatial functions such as distance, area, union, intersection, and specialty geometry data types to the database, using the OGC standards.

The bad news is that they all have the feeling of an OO add-on stuck to the side of the RDBMS model. The simple truth is that GIS is different from RDBMS. The best user interface for GIS is graphical, while RDBMS works best with the linear programming language Dr. Codd required in his famous 12 rules.

Concluding Thoughts

The advent of Internet maps (MapQuest, Google Maps, etc.) and personal GPS on cellphones has made “the man on the street” (pun intended) aware of GIS as a type of everyday data. Touch screens also have made it easy to use a GIS system on portable devices. But this hides the more complex uses that are possible when traditional data is put in a GIS database.

References

1. Gray, J. (2004). Szalay, A. S.; Fekete, G.; Mullane, W. O.; Nieto-Santisteban, M. A.; Thakar, A.R., Heber, G.; Rots, A. H.; MSR-TR-2004-32, There goes the neighborhood: Relational algebra for spatial data search. http://research.microsoft.com/pubs/64550/tr-2004-32.pdf.
2. Open Geospatial Consortium. *OpenGIS simple features specifications for SQL*. 1997; In: *http://www.opengis.org/docs/99-049.pdf*; 1997.
3. Picquet C. *Rapport sur la marche et les effets du choléra dans Paris et Le Département de la Seine*. 1832; In: *http://gallica.bnf.fr/ark:/12148/bpt6k842918*; 1832.
4. Szalay, A. (2005). Gray, J.; Fekete, G.; Kunszt, P.; Kukol, P. & Thakar A. MSR-TR-2005-123, Indexing the sphere with the hierarchical triangular mesh. http://research.microsoft.com/apps/pubs/default.aspx?id=64531.

CHAPTER 9

Big Data and Cloud Computing

Abstract

Big Data is largely a buzzword in IT right now. It was coined by Forrester Research to put a wrapper around existing database mining, data management, and other extensions of existing technology to the current hardware. The goal is to use mixed tools with larger volumes of several different forms of data being brought together under one roof. Along with this approach to data, we are also concerned with cloud computing, which is a public or private Internet network that replaces the tradition hardwired landlines within a company.

Keywords

Forrester Research; V-list; cloud computing; Big Data; data mining

Introduction

The term *Big Data* was invented by Forrester Research along with the four V buzzwords—volume, velocity, variety, variability—in a whitepaper. It has come to apply to an environment that uses a mix of the database models we have discussed so far and tries to coordinate them.

There is a "Dilbert" cartoon where the pointy-haired boss announces that "Big Data lives in the Cloud. It knows what we do" (*http://dilbert.com/strips/comic/2012-07-29/*). His level of understanding is not as bad as usual for this character. Forrester Research created a definition with the catchy buzz phrase "the four V's—volume, velocity, variety, variability" that sells a fad. Please notice value, veracity, validation, and verification are not in Forrester's V-list.

The first V is *volume*, but that is not new. We have had terabyte and petabyte SQL databases for years; just look at Wal-Mart's data warehouses. A survey in 2013 from IDC claims that the volume of data under management by the year 2020 will be 44 times greater than what was managed in 2009. But this does not mean that the data will be concentrated.

The second V is *velocity*. Data arrives faster than it has before, thanks to improved communication systems. In 2013, Austin, TX, was picked by

Google as the second U.S. city to get their fiber-optic network. I get a daily summary of my checking account transactions; only paper checks that I mailed are taking longer than an hour to clear. In the 1970s, the Federal Reserve was proud of 24-hour turnaround.

The third V is *variety*. The sources of data have increased. Anyone with a cellphone, tablet or home computer is a data source today. One of the problems we have had in the database world is that COBOL programmers came to tiered architectures with a mindset that assumes a monolithic application. The computation, data management and data presentation are all in one module. This is why SQLs still have string functions to convert temporal data into local presentation formats, to put currency symbols and punctuation in money and so forth. Today, there is no way to tell what device the data will be on, who the end user will be. We need loosely coupled modules with strong cohesion more than ever.

The fourth V is *variability*. Forrester meant this to refer to the variety of data formats. We are not using just simple structured data to get that volume. In terms of pure byte count, video is easily the largest source in my house. We gave up satellite and cable television and only watch Internet shows and DVDs. Twitter, emails, and other social network tools are also huge. Markup languages are everywhere and getting specialized. This is why ETL tools are selling so well.

Think of varying to mean both volume and velocity. Television marketing companies know that they will get a very busy switch board when they broadcast a sale. They might underestimate what the volume or velocity will be, but they know it is coming. We do not always have that luxury; a catastrophe at one point in the system can cascade. Imagine that one of your major distribution centers was in Chelyabinsk, Russia, when the meteor hit on February 13, 2013. The more centralized your system, the more damage a single event can do. If that was your only distribution center, you could be out of business.

The other mindset problem is management and administration with Big Data. If we cannot use traditional tools on Big Data, then what do we do about data modeling, database administration, data quality, data governance, and database programming?

The purpose of Big Data, or at least the sales pitch, is that we can use large amounts of data to get useful insights that will help an enterprise. Much like agile programming became an excuse for bad programming. You need to have some idea of, say, the data quality. In statistics, we say “sample size does not

overcome sample basis," or that a small random herd of cattle is better than a large, diseased herd.

In more traditional databases (small herd), people will see and clean some data, but most raw Big Data is not even eyeballed because there is simply too much of it (large herd). The lack of quality has to be huge and/or concentrated to be seen in the volume of data.

Even worse, the data is often generated from automated machinery without any human intervention. Hopefully, the data generation is cleaning the data as it goes. But one of the rules of systemantics is that fail-safe systems fail by failing to fail safely (Gall, 1977).

In fairness to Big Data, you should not assume that "traditional data" has been following best practices either. I will argue that good data practices still apply, but that they have to be adapted to the Big Data model.

9.1 Objections to Big Data and the Cloud

> *"Nothing is more difficult than to introduce a new order, because the innovator has for enemies all those who have done well under the old conditions and lukewarm defenders in those who may do well under the new." —Niccolo Machiavelli*

Old Nick was right, as usual. As with any new IT meme, there are objections to it. The objections are usually valid. We have an investment in the old equipment and want to milk it for everything we can get out of it. But more than that, our mindset is comfortable with the old terms, old abstractions, and known procedures. The classic list of objections is outlined in the following sections.

9.1.1 Cloud Computing Is a Fad

Of course it is a fad! Everything in IT starts as a fad: structured programming, RDBMS, data warehouses, and so on. The trick is to filter the hype from the good parts. While Dilbert's pointy-haired boss thinks that "If we accept Big Data into our servers, we will be saved from bankruptcy! Let us pay!," you might want to be more rational.

If so, it is a very popular fad. Your online banking, Amazon purchases, social media, eBay, or email are already in a cloud for you. Apple and Google have been keen to embrace cloud computing, affirming the idea that this is a technology revolution with longevity. Cloud computing is a developing trend,

not a passing trend.

9.1.2 Cloud Computing Is Not as Secure as In-house Data Servers

This *is* true for some shops. I did defense contract work in the Cold War era when we had lots of security in the hardware and the software. But very few of us have armed military personnel watching our disk drives. Most shops do not use encryption on everything. High security is a very different world.

But you have to develop protection tools on your side. Do not leave unencrypted data on the cloud, or on your in-house servers. When you leave it on your in-house servers, it can get hacked, too. Read about the T.J. Maxx scandal or whatever the "security breach du jour" happens to be. This will sound obvious, but do not put encryption keys in the cloud with the data it encrypts. Do not concentrate data in one site; put it in separate server locations. When one site is hacked, switch over to another site.

9.1.3 Cloud Computing Is Costly

Yes, there are initial costs in the switch to the cloud. But there are trade-offs to make up for it. You will not need a staff to handle the in-house servers. Personnel are the biggest expense in any technological field. We are back to a classic trade-off. But if you are starting a new business, it can be much cheaper to buy cloud space instead of your own hardware.

9.1.4 Cloud Computing Is Complicated

Who cares? You are probably buying it as a service from a provider. As the buyer, your job is to pick the right kind of cloud computing for your company. The goal is to keep the technical side as simple as possible for your staff. And for your users!A specialized company can afford to hire specialized personnel. This is the same reason that you buy a package that has a bunch of lawyers behind it.

9.1.5 Cloud Computing Is Meant for Big Companies

Actually, you might avoid having to obtain costly software licenses and skilled personnel when you are a small company. There are many examples of

very small companies going to the cloud so they could reach their users. If you are successful, then you can move off the cloud; if you fail, the cost of failure is minimized.

9.1.6 Changes Are Only Technical

Did the automobile simply replace the horse? No. The design of cities for automobiles is not the same as for horses. The Orson Welles classic movie *The Magnificent Ambersons* (1942) ought to be required viewing for nerds. The story is set in the period when the rise of the automobile changed American culture. It is not just technology; it is also culture.

Let me leave you with this question: How does the staff use the company internal resources with the cloud? If there is an onsite server problem, you can walk down the hall and see the hardware. If there is a cloud problem, you cannot walk down the hall and your user is still mad.

There are no purely technical changes today; the lawyers always get involved. My favorite example (for both the up and down side of cloud computing) was a site to track local high school and college sports in the midwest run by Kyle Godwin. He put his business data in Megaupload, a file-sharing site that was closed down by the Department of Justice (DOJ) for software piracy in January 2013. When he tried to get his data back, the DOJ blocked him claiming it was not his data. His case is being handled by the Electronic Freedom Foundation (EFF) as of April 2013.

There are some legal questions about the ownership of the data, so you need to be sure of your contract. Some of this has been decided for emails on servers, but it is still full of open issues.

Using the Cloud

Let me quote from an article by Pablo Valerio (2013):

> In case you need to make sure your data is properly identified as yours, and to avoid any possible dispute, the next time you negotiate an agreement with a cloud provider, you'd be wise to include these provisions in the contract:
>
> ◆ Clearly specify the process, duration, and ways the data will be returned to you, at any time in the contract duration.
>
> ◆ Also specify the format your data should be returned—usually the format the data was stored in the first place.

- Establish a limit, usually days, when the data should be fully returned to your organization.
- Clearly establish your claims of ownership of the data stored, and that you don't waive any rights on your property or copyright.
- Sometimes we just accept service agreements (where we just agree on the conditions set forth by the provider) without realizing the potential problems. I seriously recommend consulting an attorney.

9.1.7 If the Internet Goes Down, the Cloud Becomes Useless

This is true; the Internet connection is a point of vulnerability. Netflix had losses when their service, Amazon Web Services (AWS), went down on multiple occasions.

This was also true of power supplies for large data centers. National Data Corporation in Atlanta, GA, did credit card processing decades ago. Their response to their first major power failure was to route power lines from separate substations. When both substations failed during a freak ice storm, they added a third power line and a huge battery backup.

If the whole Internet is down, that might mean the end of world as we know it. But you can have a backup connection with another provider. This is a technical issue and you would need to frame it in terms of the cost to the company of an outage. For example, one of my video download sites lost part of its dubbed anime; they are the only source for the series I was watching, so I was willing to wait a day for it to come back up. But when a clothing site went down, I simply made my gift certificate order with another site.

9.2 Big Data and Data Mining

Data mining as we know it today began with data warehousing (a previous IT fad). Data warehousing concentrated summary data in a format that was more useful for statistical analysis and reporting. This lead to large volumes of data arranged in star and snowflake schema models, ROLAP, MOLAP, and other OLAP variants.

The data warehouse is denormalized, does not expect to have transactions,

and has a known data flow. But it is still structured data. Big Data is not structured and contains a variety of data types. If you can pull out structured data in the mix, then there are already tools for it.

9.2.1 Big Data for Nontraditional Analysis

More and more, governments and corporations are monitoring your tweets and Facebook posts for more complex data than simple statistical analysis. *U.S. News and World Report* ran a story in 2013 about the IRS collecting a "huge volume" of personal data about taxpayers. This new data will be mixing it with the social security numbers, credit card transactions, and the health records they will enforce under ObamaCare to create robo-audits via machines. The movie *Minority Report* (2002, Steven Spielberg, based on a Philip K. Dick short story) predicts a near future where a "precrime" police division uses mutants to arrest people for crimes they have not yet committed. You are simply assumed guilty with no further proof.

Dean Silverman, the IRS's senior advisor to the commissioner, said the IRS is going to devote time to even scrutinizing your Amazon.com purchases. This is not new; RapLeaf is a data mining company that has been caught harvesting personal data from social networks like Facebook and MySpace, in violation of user privacy agreements. Their slogan is "Real-Time Data on 80% of U.S. Emails" on their website. The gimmick is that processing unstructured data from social networks is not easy. You need a tool like IBM's Watson to read and try to understand it.

In May 2013, the Government Accounting Office (GAO) found that the IRS has serious IT security problems. They have addressed only 58 of the 118 system security-related recommendations the GAO made in previous audits. The follow-up audit found that, of those 58 "resolved" items, 13 had not been fully resolved. Right now, the IRS is not in compliance with its own policies. It is not likely that its Big Data analytics will succeed, especially when they have to start tracking ObamaCare compliance and penalizing citizens who do not buy health insurance.

In 2010, Macy's department stores were still using Excel spreadsheets to analyze customer data. In 2013, Macys.com is using tens of millions of terabytes of information every day, which include social media, store transactions, and even feeds in a system of Big Data analytics. They estimate that this is a major boost in store sales.

Kroger CEO David Dillon has called Big Data analytics his "secret

weapon" in fending off other grocery competitors. The grocery business works on fast turnaround, low profit margins, and insanely complicated inventory problems. Any small improvement is vital.

Big retail chains (Sears, Target, Macy's, Wal-Mart, etc.) want to react to market demand in near-real time. There are goals:

- Dynamic pricing and allocation as goods fall in and out of fashion. The obvious case is seasonal merchandise; Christmas trees do not sell well in July. But they need finer adjustments. For example, at what price should they sell what kind of swimwear in which part of the country in July?
- Cross-selling the customer at the cash register. This means that the customer data has to be processed no slower than a credit card swipe.
- Tighter inventory control to avoid overstocking. The challenge is to put external data such as weather reports or social media with the internal data retailers already collect. The weather report tells us *when and how many* umbrellas to send to Chicago, IL. The social media can tell us *what kind* of umbrellas we should send.

The retail chain's enemy in the online, dot-com retailers: clicks versus bricks. They use Big Data, but use it differently. Amazon.com invented the modern customer recommendation retail model. Initially, it was crude and needed tuning. My favorite personal experience was being assured that other customers who bought an obscure math book also like a particular brand of casual pants (I wear suits)! Trust me, I was the only buyer of that book on Earth that month. Today, my Netflix and Amazon recommendations are mostly for things I have already read, seen, or bought. This is means that my profiles are correctly adjusted today. When I get a recommendation I have not seen before, I have confidence in it.

9.2.2 Big Data for Systems Consolidation

The Arkansas Department of Human Services (DHS) has more than 30 discrete system silos in an aging architecture. There has been no "total view" of a client. They are trying to install a new system that will bring the state's social programs together, including Medicaid, the Supplemental Nutrition Assistance Program (SNAP), and the State Children's Health Insurance Program (SCHIP). This consolidation is going to have to cross multiple agencies, so there will be political problems as well as technical ones.

The goal is to create a single point of benefits to access all the programs

available to them and a single source to inform once all the agencies about any change of circumstances regardless of how many benefits programs the client uses.

In a similar move, the Illinois DHS decided to digitize thousands of documents and manage them in a Big Data model. In 2010, the DHS had more than 100 million pieces of paper stored in case files at local offices and warehouses throughout the state. You do not immediately put everything in the cloud; it is too costly and there is too much of it. Instead, the agency decided to start with three basic forms that deal with the applications and the chronological case records stored as PDF files. The state is using the IBM Enterprise Content Management Big Data technologies. When a customer contacts the agency, a caseworker goes through a series of questions and inputs the responses into an online form. Based on the information provided, the system determines program eligibility, assigns metadata, and stores the electronic forms in a central repository. Caseworker time spent retrieving information has gone from days to just seconds, which has been a big boost to customer service. Doug Kasamis, CIO at DHS, said “the system paid for itself in three months.”

Concluding Thoughts

A survey at the start of 2013 by Big Data cloud services provider Infochimps found that 81% of their respondents listed Big Data/advanced analytics projects as a top-five 2013 IT priority. However, respondents also report that 55% of Big Data projects do not get completed and that many others fall short of their objectives. We grab the new fad first, then prioritize business use cases. According to Gartner Research’s “Hype Cycle,” Big Data has reached its “peak of inflated expectations” by January 2013. This is exactly what happened when the IT fad du-jour was data warehouses. The failures were from the same causes, too! Overreaching scope, silos of data that could not be, and management failures.

But people trusted data warehouses because they were not exposed to the outside world. In mid-2013, we began to find out just how much surveillance the Obama administration has on Americans with the Prism program. That surveillance is done by using the cloud to monitor emails, social networks, Twitter, and almost everything else. The result has been a loss of confidence in Big Data for privacy.

References

1. Adams S. *Dilbert*. 2012; In: *http://dilbert.com/strips/comic/2012-07-29/*; 2012.
2. Gall J. *Systemantics: How systems work and especially how they fail*. Orlando, FL: Quadrangle; 1977.
3. Gartner Research. (2010–2013). http://www.gartner.com/technology/research/hype-cycles/.
4. McClatchy-Tribune Information Services. *Illinois DHS digitizes forms, leverages mainframe technology*. 2013; In: *http://cloud-computing.tmcnet.com/news/2013/03/18/6999006.htm*; 2013.
5. Satran R. *IRS High-Tech tools track your digital footprints*. 2013; In: *http://money.usnews.com/money/personal-finance/mutual-funds/articles/2013/04/04/irs-high-tech-tools-track-your-digital-footprints*; 2013.
6. Valerio P. *How to decide what (& what not) to put in the cloud*. 2012; In: *http://www.techpageone.com/technology/how-to-decide-what-what-not-to-put-in-the-cloud/#.Uhe1PdJOOSo*; 2012.
7. Valerio P. *Staking ownership of cloud data*. 2013; In: *http://www.techpageone.com/technology/staking-ownership-of-cloud-data/#.Uhe1QtJOOSo*; 2013.

CHAPTER 10

Biometrics, Fingerprints, and Specialized Databases

Abstract

Biometric databases attempt to identify people from biological and physical differences. Some (very) old methods for identifying people were branding, tattooing, and maiming; this is not popular today. This kind of data goes back to naive biometrics that start with simple photographs, then moved to Bertillon's measurement system, and finally to various fingerprinting systems. The Bertillon system evolved into machine-collected and precise measurements. Facial recognition evolved from human judgment to photographic and geometric-based computerized systems. Finally, we added retina prints and then moved to DNA to get human identity within 600 billion possibilities.

Keywords

Bertillon card; biometrics; DNA (deoxyribonucleic acid); facial recognition; fingerprints; Galton system; loop; whorls; arch; NIST (National Institute for Science and Technology); SO/IEC 19794-14:2013: Information Technology —Biometric Data Interchange Formats; STR (short tandem repeats); VNTR (variable-number tandem repeats)

Introduction

Currently, biometrics fall outside commercial use. They identify a person as a *biological entity* rather than a commercial entity. We are now in the worlds of medicine and law enforcement. Eventually, however, biometrics may move into the commercial world as security becomes an issue and we are willing to trade privacy for security.

Automobiles come with VINs (vehicle identification numbers), books come with an International Standard Book Number (ISBN), companies have a Data Universal Numbering System (DUNS), and retail goods come with a Universal Product Code (UPC) bar code. But people are not manufactured goods and do not have an ANSI/ISO standard.

One of the standard troll questions on SQL database forums is how to

identify a person with a natural key. The troll will object to any identifier that is linked to a role a person plays in the particular data model, such as an account number. It has to be a magical, universal "person identifier" that is 100% error-free, cheap to use, and instantly available. Of course, pointing out that you do not have a magical, universal generic database for all of humanity, so it is both absurd and probably illegal under privacy laws, does not stop a troll.

We are biological goods! We do have biometric identifiers by virtue of being biological. The first problem is collecting the raw data, the biometric measurements. It obviously involves having a human body and some kind of instrument. The second problem is encoding those measurements in such a way that we can put them into a database and search them.

People are complex objects with lots of dimensions. There is no obvious way you classify a person, no common scales. You might describe someone as "he/she looks < ethnic/gender/age group>" when trying to help someone find a person in a crowd. If the person is statistically unusual and the crowd is small, this can work. Wilt Chamberlain entitled his autobiography *Wilt: Just Like Any Other 7-Foot Black Millionaire Who Lives Next Door*, as if he would be lost in a crowd. But at the other extreme, there is an advantage for fashion models to be racially and age ambiguous.

10.1 Naive Biometrics

The first biometric identifier is facial recognition. Human beings can identify other human beings by sight, but computerizing this is difficult. People use fuzzy logic in their brains and do not have precise units of measurement to classify a person. This is physically hardwired into the human brain. If the part of the brain that does this survival task is damaged, you become "face blind" and literally cannot match a photograph to the person sitting next to you, or even your own image in a mirror.

Some (very) precomputer-era methods for identifying people were branding, tattooing, and maiming to physically mark a criminal or member of some group. Think of this as a bar code solution with really bad equipment. Later, we depended on visual memory and books full of photographs. But people change over time. Body weight varies, facial hair changes with age and fashion, and age takes its toll. Matching old school yearbook photographs and current photographs of celebrities is a popular magazine feature. Yet, we still accept an awful driver's license photo as valid identification.

What we need for a database is an encoding system as opposed to a human narrative. Around 1870, French anthropologist Alphonse Bertillon devised a three-part identification system. The first part was a record of the dimensions of certain bony parts of the body. These measurements were reduced to a formula that, theoretically, would apply to *only one person* and would *not change* during his or her adult life. The other two parts were a formalized description and the "mugshot" photograph we still use today (Figure 10.1).

FIGURE 10.1 Bertillon Identification System.

This system also introduced the idea of keeping data on cards, known as Bertillon cards, that could be sorted by characteristics and retrieved quickly instead of paper dossiers. A trained, experienced user could reduce hundreds of thousands of cards down to a small deck of candidates that a human could compare against a suspect or photograph. The cards used holes on the edges

to make visual sorting easier.

The Bertillon card encoded the prisoner's eyes, ears, lips, beard, hair color, skin color, ethnicity, forehead, nose, build, chin, general contour of head, hair growth pattern, eyebrows, eyeballs and orbits, mouth, physiognomic expression, neck, inclination of shoulders, attitude, general demeanor, voice and language, and clothing.

The Bertillon system was generally accepted for over 30 years. Since you had to have the person to measure him or her, and it takes a while, it was used to determine if a suspect in custody was a repeat offender or repeat suspect. It was not useful for crime scene investigations (CSI) that we see on television shows.

The Bertillon system's descendants are the basis for facial recognition systems, hand geometry recognition, and other biometric identification systems. Rather than trying to reduce a person to a single number, modern systems are based on ratios that can be constructed from still images or video.

10.2 Fingerprints

Fingerprints have been used for identification going back to Babylon and clay tablets. They were used in ancient China, Japan, and India as a signature for contracts. But it was not until 1788 when Johann Christoph Andreas Mayer, a German anatomist, recognized that fingerprints are individually unique.

Collecting fingerprints is much easier than a Bertillon card. Even today, with computerized storage, law enforcement uses the term *ten-card* for the form that holds prints from all ten fingers. As a database person would expect, the problem was the lack of a classification system. There were several options depending on where you lived. The most popular ten-print classification systems include the Roscher system, the Juan Vucetich system, and the Henry system. The Roscher system was developed in Germany and implemented in both Germany and Japan. The Vucetich system was developed in Argentina and is still in use in South America today. The Henry system was developed in India and implemented in most English-speaking countries, including the United States. Today, it is usually called the Galton–Henry classification because of the work done by Sir Francis Galton from which the Henry system was built.

10.2.1 Classification

In the original Galton system of classification, there are three basic fingerprint patterns: loop (60–65%), whorl (30–35%), and arch (5%). From this basic model, we get more subclassifications for plain arches or tented arches, and into loops that may be radial or ulnar, depending on the side of the hand toward which the tail points. Ulnar loops start on the pinky-side of the finger, the side closer to the ulna, the lower arm bone. Radial loops start on the thumb-side of the finger, the side closer to the radius. Whorls may also have subgroup classifications including plain whorls, accidental whorls, double-loop whorls, peacock's eye, composite, and central pocket–loop whorls. Then there are tented arch, the plain arch, and the central pocket loop.

The modern system should delight a database person because it uses a simple hashing algorithm. It consists of five fractions, in which *R* stands for right, *L* for left, *i* for the index finger, *m* for the middle finger, *t* for the thumb, *r* for the ring finger, and *p* (pinky) for the little finger. The encoding is *Ri*/*Rt* + *Rr*/*Rm* + *Lt*/*Rp* + *Lm*/*Li* + *Lp*/*Lr* The numbers assigned to each print are based on whether or not they are whorls. A whorl in the first fraction is given a 16, the second an 8, the third a 4, the fourth a 2, and 0 to the last fraction. Arches and loops are assigned 0. The numbers in the numerator and denominator are added up, using the scheme:

$$(Ri + Rr + Lt + Lm + Lp)/(Rt + Rm + Rp + Li + Lr)$$

and 1 is added to both top and bottom, to exclude any possibility of division by 0. For example, if the right ring finger and the left index finger have whorls, the encoding would look like this:

$$0/0 + 8/0 + 0/0 + 0/2 + 0/0 + 1/1, \text{ and the calculation: } (0 + 8 + 0 + 0 + 0 + 1)/(0 + 0 + 0 + 2 + 0 + 1) = 9/3 = 3$$

Only fingerprints with a hash value of 3 can match this person.

10.2.2 Matching

Matching a fingerprint is not the same as classifying it. The ten-card is made by rolling each finger on the ten-card to get the sides of the finger as well. In the real world, the sample to be matched against the database is a partial print, smeared or damaged.

This meant a search had to start with the classification as the first filter, then the technician counted the ridges. The final matches were done by hand. At one time, IBM made a special device with a front panel that had ten rotary dial switches, one for each finger with the ridge counts. This was easy for

noncomputer police personnel to operate.

Fingerprint image systems today use different technology—optical, ultrasonic, capacitive, or thermal—to measure the physical difference between ridges and valleys. The machinery can be grouped into two major families: solid-state fingerprint readers and optical fingerprint readers. The real problem is that people are soft, so each image is distorted by pressure, skin condition, temperature, and other sampling noises.

To overcome these problems we now use noncontact three-dimensional fingerprint scanners. The images are now digital. We are very good at high-resolution image processing today that can be adjusted by comparing the distances between ridges to get a scale, and distorted back to the original shape.

Since fingerprints are used for security and access, these systems typically use a template that was previously stored and a candidate fingerprint. The algorithm finds one or more points in the fingerprint image and aligns the candidate fingerprint image to it. This information is local to the hardware and it is not meant for data exchange. Big Data databases take time for matching, so we try to optimize things with algorithms and database hardware; current hardware matches around 1,000 fingerprints per second.

In April 2013 Safe Gun Technology (SGTi) said it is hoping it can begin production on its version of a smart gun within the next two months. The Columbus, GA–based company uses relatively simple fingerprint recognition through a flat, infrared reader positioned on the weapon's grip. The biometrics reader enables three other physical mechanisms that control the trigger, the firing pin, and the gun hammer. The controller chip can save from 15,000 to 20,000 fingerprints. If a large military unit wanted to program thousands of fingerprints into a single weapon, it would be possible. A single gun owner could also temporarily program a friend's or family member's print into the gun to go target shooting and then remove it upon returning home.

10.2.3 NIST Standards

NIST (National Institute for Science and Technology) has been setting standards as a member of both ANSI and ISO for decades. They deal with more than just fingerprints; they have specifications for fingerprints, palm prints, plantars, faces, irises, and other body parts, as well as scars, marks, and tattoos (SMTs). Marks, as used in this standard, mean needle marks or tracks from drug use.

The first version of this standard was ANSI/NBS-ICST 1-1986 and was a fingerprint standard. It evolved over time, with revisions made in 1993, 1997, 2000, and 2007. In 2008, NIEM-conformant encoding using eXtensible Markup Language (XML) was adopted. NIEM (National Information Exchange Model) is a partnership of the U.S. Department of Justice and Department of Homeland Security. The most useful part of this document for the causal user is the list of the types of identifiers and their records, shown in Table 10.1.

Table 10.1

Types of Identifiers and Their Records

Record Identifier	Record Contents
1	Transaction information
2	User-defined descriptive text
3	Low-resolution grayscale fingerprint image (deprecated)
4	High-resolution grayscale fingerprint image
5	Low-resolution binary fingerprint image (deprecated)
6	High-resolution binary fingerprint image (deprecated)
7	User-defined image
8	Signature image
9	Minutiae data
10	Face, other body part, or SMT image
11	Voice data (future addition to the standard)
12	Dental record data (future addition to the standard)
13	Variable-resolution latent friction ridge image
14	Variable-resolution fingerprint image
15	Variable-resolution palm print image
16	User-defined variable-resolution testing image
17	Iris image
18	DNA data
19	Variable-resolution plantar image
20	Source representation
21	Associated context
22–97	Reserved for future use
98	Information assurance
99	Common Biometric Exchange Formats Framework (CBEFF) biometric data record

Please note that voice prints and dental records are not part of these specifications. They show up on television crime shows, but are actually so

rare they are not worth adding. Dental records are used after death in most cases to identify a corpse. Voice is hard to match and we do not have an existing database to search.

Type-4 records are single fingerprint images at a nominal scanning resolution of 500 pixels per inch (ppi). You need 14 type-4 records to have the classic ten-card in a file (ten full, rolled individual fingers, two thumb impressions, and two simultaneous impressions of the four fingers on each hand). We want to move over to the type-14 records for fingerprint images.

Type-18 records are for DNA. This uses the ISO/IEC 19794-14:2013 Information Technology—Biometric Data Interchange Formats—Part 14: DNA Data Standard. Because of privacy considerations, this standard only uses the noncoding regions of DNA and avoids phenotypic information in other regions. More on DNA in the next section.

As expected, we have lots of three-letter acronyms: SAP (subject acquisition profile) is the term for a set of biometric characteristics. These profiles have mnemonics: SAP for face, FAP for fingerprints, and IAP for iris records. SAP codes are mandatory in type-10 records with a face image; FAP is optional in type-14 records; and IAP is optional in type-17 records.

Moving to machine processing is important. Humans are simply too error-prone and slow for large databases. In 1995, the Collaborative Testing Service (CTS) administered a proficiency test that, for the first time, was “designed, assembled, and reviewed” by the International Association for Identification (IAI) to see how well trained personnel did with actual data.

The results were disappointing. Four suspect cards with prints of all ten fingers were provided together with seven latent prints. Of 156 people taking the test, only 68 (44%) correctly classified all seven latent prints. Overall, the tests contained a total of 48 incorrect identifications. David Grieve, the editor of the *Journal of Forensic Identification*, describes the reaction of the forensic community to the results of the CTS test as ranging from “shock to disbelief,” and added:

> *Errors of this magnitude within a discipline singularly admired and respected for its touted absolute certainty as an identification process have produced chilling and mind-numbing realities. Thirty-four participants, an incredible 22% of those involved, substituted presumed but false certainty for truth. By any measure, this represents a profile of practice that is unacceptable and thus demands positive action by the entire community.*

10.3 DNA Identification

DNA (deoxyribonucleic acid) profiling is *not* full genome sequencing. Profiling is complete enough for paternity testing and criminal evidence. The good news is that a DNA profile can be encoded as a set of numbers that can be used as the person's identifier. Although 99.9% of human DNA sequences are the same in every person, and humans and chimpanzees differ by less than 2%, there are enough differences to distinguish one individual from another.

In the case of monozygotic ("identical") twins, there is still enough differences that they can be differentiated by going to the gene level (*http://www.scientificamerican.com/article.cfm?id=identical-twins-genes-are-not-identical*). At sites of genetic divergence, one twin would have a different number of copies of the same gene, a genetic state called *copy number variants*. Normally people carry two copies of every gene, one inherited from each parent. But there are regions in the genome that can carry anywhere from 0 to over 14 copies of a gene.

Most of us are familiar with paternity testing that we have seen on television or read about in the tabloids. The initial testing that is done can quickly rule out a father from a sample from the mother and the child. Table 10.2 is a sample report from a commercial DNA paternity test that uses five markers.

Table 10.2

Sample Paternity Report

DNA Marker	Mother	Child	Alleged Father
D21S11	28, 30	28, 31	29, 31
D7S820	9, 10	10, 11	11, 12
TH01	14, 15	14, 16	15, 16
D13S317	7, 8	7, 9	8, 9
D19S433	14, 16.2	14, 15	15, 17

The alleged father's DNA matches among these five markers, so he is looking guilty. The complete test results need to show this matching on 16 markers between the child and the alleged father to draw a conclusion of whether or not the man is the biological father. The initial test might spot close male relatives. For humans, the complete genome contains about 20,000 genes on 23 pairs of chromosomes. Mapping them is expensive and time consuming.

10.3.1 Basic Principles and Technology

DNA profiling uses repetitive ("repeat") sequences that are highly variable, called *variable number tandem repeats* (VNTRs), particularly short tandem repeats (STRs). VNTR loci are very similar between closely related humans, but so variable that unrelated individuals are extremely unlikely to have the same VNTRs.

This DNA profiling technique was first reported in 1984 by Sir Alec Jeffreys at the University of Leicester in England, and is now the basis of several national DNA databases. Dr. Jeffreys' genetic fingerprinting was made commercially available in 1987, when a chemical company, Imperial Chemical Industries (ICI), started a blood-testing center in England.

The goal has been personalized medicine rather than identification. Identification does not need a full genome, so it should be cheaper. To get an idea of the cost reduction, the first complete sequencing of a human genome, done by the Human Genome Project, cost about $3 billion when it was finally completed in 2003. As of 2010, we can identify markers for specific diseases and genetic traits for under $1,000. Dogs can be identified for under $100; this has been used by anal (pun intended) homeowner associations to tag dogs and their poop to fine owners who do not clean up after their animals.

While every country uses STR-based DNA profiling systems, they do not all use the same one. In North America, CODIS 13 core loci are almost universal, while the United Kingdom uses the SGM + 11 loci system (which is compatible with their national DNA database). There are overlaps in the sets of STR regions used because several STR regions can be tested at the same time.

Today, we have microchips from several companies (Panasonic, IBM, Fujitsu, etc.) that can do a DNA profile in less than an hour. The chips are the size of a coin and work with a single drop of blood or other body fluid. DNA is extracted inside the chip via submicroscopic "nanopores" (holes) in the chip from the blood. A series of polymerase chain reactions (PCRs) are completed inside the chip, which can be read via an interface.

10.4 Facial Databases

We have grown up with television crime shows where a video surveillance camera catches the bad guy, and the hero takes a freeze frame from the video back to the lab to match it against a facial database of bad guys. The mugshots flash on a giant television screen so fast you cannot recognize

anyone, until a single perfect match pops up and the plot advances. The video graphics are beautiful and flashy, the heroes are beautiful and flashy, too. The case is always solved, unless we need a cliffhanger for the next season.

Facial databases do not work that way. Like any other biometric data, it can be used in a security system with a local database or as part of a centralized database. While it is pretty much the same technology, database people do not care so much about personal security uses.

Recognition algorithms can be divided into two main approaches: geometric or photometric. *Photometric* algorithms basically try to overlay the pictures for a match. It easy to have multiple matches when you have a large database to search. This approach automates what people do by eyeball. There is a section of the human brain that does nothing but recognize faces, so we have evolved a complex process for this survival trait ("Hey, you're not my tribe!"). However, some people have a brain condition called prosopagnosia, or "face blindness," and they cannot do this. But I digress.

Geometric algorithms extract landmarks, or features, from an image of the subject's face. These points, such as the center of the eyes, top of the nose, cheekbones, and so forth, can be normalized and then compressed to a mathematical value. A "probe image" is compared with the compressed face data. This is like the hashing algorithm or CRC code—you get a set of candidate matches that you filter.

There is no single algorithm, but some of them are principal component analysis using eigenfaces, linear discriminate analysis, elastic bunch graph matching using the Fisherface algorithm, hidden Markov model, multilinear subspace learning using tensor representation, and neuronal motivated dynamic link matching, if you want to read more technical details.

The bad news is that faces are three dimensional, not flat like fingerprints. In older methods, the viewing angle, lighting, masks, hats, and hairdos created enough "noise-to-signal" that those wonderful matches you see on television shows are not always possible. They worked from static, flattened images from the 8 × 10 glossy headshots of actors!

10.4.1 History

The earliest work in this area was done for an intelligence agency in the mid-1960s by Woody Bledsoe, Helen Chan Wolf, and Charles Bisson. Using an early graphics tablet, an operator would mark coordinates and 20 distances on a mugshot. With practice, they could do about 40 photographs per house. The

recognition algorithm was a simple match of the set of distances for the suspect and the database records. The closest matches were returned, using a chi-square matching algorithm.

The real problem was normalizing the data to put the face into a standard frontal orientation. The program had to determine the tilt, lean, and rotation angles, then use projective geometry to make adjustments. To quote Bledsoe (1966): “In particular, the correlation is very low between two pictures of the same person with two different head rotations.” The normalization assumed a “standard head” to which the points and distance could be assigned. This standard head was derived from measurements on seven real heads.

Today, products use about 80 nodal points and more sophisticated algorithms. Some of these measured by the software are the:

- Distance between the eyes
- Width of the nose
- Depth of the eye sockets
- Shape of the cheekbones
- Length of the jaw line

There is no standard yet, but Identix, a commercial company, has a product called FaceIt®. This product can produce a “face print” from a three-dimensional image. FaceIt currently uses three different templates to confirm or identify the subject: vector, local feature analysis, and surface texture analysis. This can then be compared to a two-dimensional image by choosing three specific points off of the three-dimensional image. The face print can be stored in a computer as numeric values. They now use the uniqueness of skin texture, to yield even more accurate results.

That process, called surface texture analysis (STA), works much the same way facial recognition does. A picture is taken of a patch of skin, called a skin print. That patch is then broken up into smaller blocks. This is not just skin color, but any lines, pores, and the actual skin texture. It can identify differences between identical twins, which is not yet possible using facial recognition software alone.

The vector template is very small and is used for rapid searching over the entire database primarily for one-to-many searching. Think of it as a kind of high-level index on the face. The local feature analysis (LFA) template performs a secondary search of ordered matches following the vector template. Think of it as a secondary-level index, after the gross filtering is

done.

The STA is the largest of the three. It performs a final pass after the LFA template search, relying on the skin features in the image, which contain the most detailed information.

By combining all three templates, FaceIt is relatively insensitive to changes in expression, including blinking, frowning, or smiling, and has the ability to compensate for mustache or beard growth and the appearance of eyeglasses. The system is also uniform with respect to race and gender.

Today, sensors can capture the *shape* of a face and its features. The contour of the eye sockets, nose, and chin can be unique in an individual. Think of Shrek; he is really a three-dimensional triangular mesh frame with a skin over it. This framework can be rotated and flexed and yet still remain recognizable as Shrek. *Very recognizable as Shrek.* But this requires sophisticated sensors to do face capture for the database or the probe image. Skin texture analysis is a more recent tool that works with the visual details of the skin, as captured in standard digital or scanned images. This can give a 20–25% performance improvement in recognition.

10.4.2 Who Is Using Facial Databases

The first thought is that such databases are only for casinos looking for cheats and criminals or government police agencies. Yes, there is some of that, but there are mundane commercial applications, too.

Google's Picasa digital image organizer has a built-in face recognition system starting in version 3.5 onward. It can associate faces with persons, so that queries can be run on pictures to return all pictures with a specific group of people together.

Sony's Picture Motion Browser (PMB) analyzes photos, associates photos with identical faces so that they can be tagged accordingly, and differentiates between photos with one person, many persons, and nobody.

Windows Live Photo Gallery also includes face recognition.

Recognition systems are also used by casinos to catch card counters and other blacklisted individuals.

Police applications are not limited to just mugshot databases in investigations:

- The London Borough of Newham tried a facial recognition system in their

borough-wide CCTV system.

- The German Federal Criminal Police Office has used facial recognition on mugshot images for all German police agencies since 2006. The European Union has such systems in place on a voluntary basis for automated border controls by Germany and Austria at international airports and other border crossing points. Their systems compare the face of the individual with the image in the e-passport microchip.
- Law enforcement agencies at the state and federal levels in the United States use arrest mugshot databases. The U.S. Department of State operates one of the largest face recognition systems in the world, with over 75 million photographs, that is actively used for visa processing.
- The US-VISIT (U.S. Visitor and Immigrant Status Indicator Technology) is aimed at foreign travelers gaining entry to the United States. When a foreign traveler receives his or her visa, he or she submits fingerprints and has his or her photograph taken. The fingerprints and photograph are checked against a database of known criminals and suspected terrorists. When the traveler arrives in the United States at the port of entry, those same fingerprints and photographs are used to verify that the person who received the visa is the same person attempting to gain entry.
- Mexico used facial recognition to prevent voter fraud in their 2000 presidential election. It was a way to detect double voting.
- At Super Bowl XXXV in January 2001, police in Tampa Bay, FL, used Viisage facial recognition software to search for potential criminals and terrorists in attendance at the event. Nineteen people with minor criminal records were potentially identified.

Facial recognition for ATM machines and personal computers has been tested, but not widely deployed. The android cell phones have an application called Visidon Applock. This application allows you to put a facial recognition lock on any of your applications. Facial recognition technology is already implemented in the iPhoto application for Macintosh. Another proposal is a smartphone application for people with prosopagnosia, so they can recognize their acquaintances. Smart cameras can detect not just focus, but closed eyes, red eyes, and other situations that mess up portrait photographs.

10.4.3 How Good Is It?

Bluntly, this is not the strongest biometric. DNA and fingerprinting are more reliable, efficient, and easier to search. The main advantage is that it does not require consent or physical samples from the subject and can find them in crowd. Ralph Gross, a researcher at the Carnegie Mellon Robotics Institute, describes one obstacle related to the viewing angle of the face: “Face recognition has been getting pretty good at full frontal faces and 20 degrees off, but as soon as you go towards profile, there have been problems.” In fact, if you have a clear frontal view of a suspect, you can find him or her in more than a million mugshots 92% of the time.

The London Borough of Newham CCTV system mentioned earlier has never recognized a single criminal, despite several criminals in the system’s database living in the Borough, after all these years. But the effect of having cameras everywhere has reduced crime. The same effect occurred in Tampa, FL, and a system at Boston’s Logan Airport was shut down after failing to make any matches during a two-year test period.

The television show *Person of Interest* that premiered in 2012 is based on the premise that our heroes have a super AI program that can hack every computer, every surveillance, and, with super AI, figure out if someone is going to be in trouble so they can save them before the end of the show. In 2012, the FBI launched a $1 billion facial recognition program called the Next-Generation Identification (NGI) project as a pilot in several states.

The FBI’s goal is to build a database of over 12 million mugshots, voice prints, and iris scans from federal criminal records and the U.S. State Department’s passport and visa databases. They can also add the DMV information from the 30-odd states that currently keep this data.

We are not at “television fantasy level” yet and probably won’t be for many years. However, a test suite provided by NIST called the Face Recognition Grand Challenge (FRGC) (*http://www.nist.gov/itl/iad/ig/frgc.cfm*) ran from May 2004 to March 2006. The newer algorithms were ten times more accurate than the face recognition algorithms of 2002 and 100 times more accurate than those of 1995. Some of them were able to recognize identical twins who fooled humans.

The 2013 Boston Marathon bombers were not spotted by a fancy FBI facial recognition system, even though Dzhokhar and Tamerlan Tsarnaev’s images existed in official databases. The images that the FBI had of the Tsarnaevs brothers were grainy surveillance camera images taken from far away, with them wearing hats and sunglasses. They were spotted by investigators seeing them planting the bombs on the surveillance footage, then matching a face

with an image from the security camera of a 7-11 in Cambridge. After the photos were made public, Facebook and hundreds of cellphone cameras filled in the gaps.

Concluding Thoughts

Biometrics will overcome the technological problems in the near future. It will be possible to have a cheap device that can match DNA and fingerprints or even put it on a smartcard. Identification at a distance is also coming. The real issues are political rather than technical.

References

1. Grieve D. *Fingerprints: Scientific proof or just a matter of opinion?* 2005; *www.sundayherald.com*; 2005; http://www.zoominfo.com/CachedPage/?archive_id=0&page_id=1324386827&page_url=//www.sundayherald 11-21T04:47:08&firstName=David&lastName=Grieve.
2. Gross, R. (2001); Shi, J.; Cohn, J. F. *Quo vadis face recognition?* http://www.ri.cmu.edu/pub_files/pub3/gross_ralph_2001_1/gross_ral
3. Bledsoe WW. *Semiautomatic facial recognition*. Technical Report SRI Project 6693 Menlo Park, CA: Stanford Research Institute; 1968.
4. Bledsoe WW. *Man-machine facial recognition: Report on a large-scale experiment*. Technical Report PRI 22 Palo Alto, CA: Panoramic Research, Inc; 1966.
5. Bledsoe WW. *The model method in facial recognition*. Technical Report PRI 15 Palo Alto, CA: Panoramic Research, Inc; 1964.

CHAPTER 11

Analytic Databases

Abstract

Analytic databases were created to do analysis as opposed to simple data retrieval and aggregation that we have with a traditional SQL database used for OLTP. This model uses different structures for data, such as cubes and rollups. The theoretical foundation of OLAP came from the same Dr. Codd that gave use RDBMS. Dr. Codd described four analysis models in his whitepaper: (1) categorical—this is the typical descriptive statistics we have had in reports since the beginning of data processing; (2) exegetical—this is what we have been doing with spreadsheets, slice, dice, and drilldown reporting on demand; (3) contemplative—this is "what if" analysis; and (4) formulaic—these are goal-seeking models. You know what outcome you want, but not how to get it.

Keywords

Dr. E. F. Codd; cube; HOLAP; MOLAP; OLAP (online analytical processing); OLTP (online transaction processing); ROLAP

Introduction

The traditional SQL database is used for online transaction processing (OLTP). Its purpose is to provide support for daily business applications. The hardware was too expensive to keep a machine for special purposes, like analyzing the data. That was then; this is now. Today, we have online analytical processing (OLAP) databases, which are built on the OLTP data.

These products use a snapshot of a database taken at one point in time, and then put the data into a dimensional model. The purpose of this model is to run queries that deal with aggregations of data rather than individual transactions. It is analytical, not transactional.

In traditional file systems and databases, we use indexes, hashing, and other tricks for data access. We still have those tools, but we have added new ones. Star schemas, snowflake schemas, and multidimensional storage methods are all ways to get to the data faster, but in the aggregate rather than by rows.

11.1 Cubes

One such structure is the cube (or hypercube). Think of a two-dimensional cross-tabulation or a spreadsheet that models, say, a location (encoded as north, south, east, and west) and a product (encoded with category names). You have a grid that shows all possible combinations of locations and products, but many of the cells are going to be empty—you do not sell many fur coats in the south, or many bikinis in the north.

Now extend the idea to more and more dimensions, such as payment method, purchase time, and so forth; the grid becomes a cube, then a hypercube, and so forth. If you have trouble visualizing more than three dimensions, then imagine a control panel like you find on a stereo system. Each slider switch controls one aspect of the sound, such as balance, volume, bass, and treble. The dimensions are conceptually independent of each other, but define the whole.

As you can see, actually materializing a complete cube would be very expensive and most of the cells would be empty. Fortunately, we have prior experience with sparse arrays from scientific programming and a lot of database access methods.

The OLAP cube is created from a star schema of tables, which we will discuss shortly. At the center is the fact table that lists the core facts that make up the query. Basically, a star schema has a fact table that models the cells of a sparse array by linking them to dimension tables. The star is denormalized, but since the data is never updated, there is no way to get an anomaly and no need for locking. The fact table has rows that store a complete event, such as a purchase (who, what, when, how, etc.). while the dimensional tables provide the units of measurement (e.g., the purchase can be grouped into weekday, year, month, shopping season, etc.).

11.2 Dr. Codd's OLAP Rules

Dr. E. F. Codd and Associates published a whitepaper for Hyperion Solutions (see Arbor Software) in 1993 entitled "Providing OLAP to User-Analysts: An IT Mandate." This introduced the term OLAP and a set of abstract rules somewhat like his rules for RDBMS. However, because the paper had been sponsored by a commercial vendor (whose product matched those rules fairly well), rather than a pure research paper like his RDBMS work, it was not well received.

It was also charged that Dr. Codd himself allowed his name to be used and

that he did not put much work into it, but let the vendor, his wife, and a research assistant do the work. The original whitepaper had 12 rules, and then added another 6 rules in 1995. The rules were restructured into four feature groups, which are summarized here.

In defense of Dr. Codd, his approach to first defining a new technology in terms of easily understood abstractions and generalizations is how every database innovator who followed him has approached their first public papers. It is only afterward that you will see scholarly mathematical papers that nobody but an expert can understand. His ideas on OLAP have also stood up over time.

11.2.1 Dr. Codd's Basic Features

Dr. Codd's original numbering is retained here, but some more comments are included.

- *F1: Multidimensional conceptual view.* This means that data is kept in a matrix in which each dimension of the matrix is an attribute. This is a grid or spreadsheet model of the data. This includes the ability to select subsets of data based on restrictions on the dimensions.
- *F2: Intuitive data manipulation.* Intuition is a vague term that every vendor claims for their product. This is usually taken to mean that that you have a graphical user interface (GUI) with the usual drag-and-drop and other graphic interfaces. This does not exclude a written programming language, but it does not give you any help with the design of it.
- *F3: Accessibility: OLAP as a mediator.* The OLAP engine is middleware between possibly heterogeneous data sources and an OLAP front end. You might want to compare this to the model for SQL, where the SQL engine sits between the user and the database.
- *F4: Batch extraction versus interpretive.* The OLAP has to have its own staging database for OLAP data, as well as offering live access to external data. This is HOLAP (hybrid OLAP), which we will discuss shortly. The live access to external data is a serious problem because it implies some kind of connection and possibly huge and unexpected data flows into the staging database.
- *F5: OLAP analysis models.* Dr. Codd described four analysis models in his whitepaper:

    1. *Categorical:* This is the typical descriptive statistics we have had in

reports since the beginning of data processing.

2. *Exegetical:* This is what we have been doing with spreadsheets—slice, dice, and drill down reporting on demand.

3. *Contemplative:* This is "what if" analysis. There have been some specialized tools for doing this kind of modeling, and some of them use extensions to spreadsheets. Contemplative analysis lets you ask questions about the effects of changes on the whole system, such as "What is the effect of closing the Alaska store do to the company?" In other words, you have a particular idea you want to test.

4. *Formulaic:* These are goal-seeking models. You know what outcome you want, but not how to get it. The model keeps changing parameters and doing the contemplations until it gets to the desired results (or proves that the goal is impossible). Here you would set a goal, such as "How can I increase the sale of bikinis in the Alaska store?" and wait for an answer. The bad news is that there can be many solutions, no solution ("Bikinis in Alaska are doomed to failure"), or unacceptable solutions ("Close down all but the Alaska store").

Categorical and exegetical features are easy to implement. The contemplative and formulaic features are harder and more expensive to implement. We have some experience with formulaic and contemplative analysis with linear or constraint programming for industrial processes. But the number of parameters had to be fairly small, very well controlled, and the results measurable in well-defined units.

This weakness has led to "fuzzy" logic and math models where the data does not have to be precise in a traditional sense and a response on a large data set can be made fast ("Leopard prints are selling unusually well in Alaska, so it might be worth sending leopard print bikinis there this summer").

- *F6: Client server architecture.* This pretty well speaks for itself. The goal is to allow the users to share data easily and be able to use any front-end tool. Today, this would be cloud-based access.

- *F7: Transparency.* This is part of the RDBMS model. The client front end should not have to be aware of how the connection to the OLAP engine or other data sources is made.

- *F8: Multi-user support.* This is also part of the RDBMS model. This is actually easy to do because OLAP engines are read-only snapshots of

data. There is no need for transaction controls for multiple users. However, there is a new breed of analytical database that is designed to allow querying while data is being steamed from external data sources in real time.

11.2.2 Special Features

This feature list was added to make the OLAP engines practical.

- *F9: Treatment of non-normalized data.* This means we can load data from non-RDBMS sources. The SQL-99 standard, part 9, also added the SQL/MED (management of external data) feature for importing external data (CAN/CSA-ISO/IEC 9075-9-02, March 6, 2003, adopted ISO/IEC 9075-9:2001, first edition, May 15, 2001). This proposal never got very far, but current ETL products can handle these transformations in their proprietary syntax.
- *F10: Store OLAP results.* This is actually a practical consideration. OLAP data is expensive and you do not want to have to reconstruct over and over from live data. Again, the implication is that the OLAP database is a snapshot of the state of the data sources.
- *F11: Extraction of missing values.* In his relational model version 2 (RMV2), Codd defined two kinds of missing values rather than the single `NULL` used in SQL. One of them is like the SQL `NULL`, which models that the attribute exists in the entity, but we do not know its value. The second kind of missing value says that the attribute does not exist in the entity, so it will never have a value. Since most SQL products (the exception is First SQL) support only the first kind of `NULL`, it can be hard to meet rule F11. However, there is some support in the `CUBE` and `ROLLUP` features for determining which `NULL`s were in the original data and which were created in the aggregation.
- *F12: Treatment of missing values.* All missing values are ignored by the OLAP analyzer regardless of their source. This follows the rules for dropping `NULL`s in aggregations in standard SQL. But in statistics, there are ways to work around missing data. For example, if the known values take the shape of a normal distribution, the system can make the assumption that the missing values are in that normal distribution.

11.2.3 Reporting Features

The reporting features are obviously the whole point of OLAP and had to be added. But this feels more commercial than theoretical.

- *F13: Flexible reporting.* This feature is again a bit vague. Let's take it to mean that the dimensions can be aggregated and arranged pretty much anyway the user wants to see the data. Today, this usually means the ability to provide pretty graphics. Visualization has become a separate area of IT.
- *F14: Uniform reporting performance.* Dr. Codd required that reporting performance would not be significantly degraded by increasing the number of dimensions or database size. This is more of a product design goal than an abstract principle. If you have a precalculated database, the number of dimensions is not so much of a problem.
- *F15: Automatic adjustment of physical level.* Dr. Codd required that the OLAP system adjust its physical storage automatically. This can be done with utility programs in most products, so the user has some control over the adjustments.

11.2.4 Dimension Control

These features deal with the dimensions on the cubes and we can use them together.

- *F16: Generic dimensionality.* Dr. Codd took the purist view that each dimension must be equivalent in both its structure and operational capabilities. This may not be unconnected with the fact that this is an Essbase characteristic. However, he did allow additional operational capabilities to be granted to selected dimensions (presumably including time), but he insisted that such additional functions should be grantable to any dimension. He did not want the basic data structures, formulas, or reporting formats to be biased toward any one dimension.
  This has proven to be one of the most controversial of all the original 12 rules (it was renumbered when the features were revised). Technology-focused products tend to largely comply with it, so the vendors of such products support it. Application-focused products usually make no effort to comply, and their vendors bitterly attack the rule. With a strictly purist interpretation, few products fully comply. I would suggest that if you are purchasing a tool for general purpose or multiple application use, then you want to consider this rule, but even then with a lower priority. If you are buying a product for a specific application, you may safely ignore this

rule.

- *F17: Unlimited dimensions and aggregation levels.* This is physically impossible, so we can settle for a "large number" of dimensions and aggregation levels. Dr. Codd suggested that the product should support at least 15 and preferably 20 dimensions. The rule of thumb is to have more than you need right now so there is room for growth.
- *F18: Unrestricted cross-dimensional operations.* There is a difference between a calculation and an operation. Certain combinations of scales cannot be used in the same calculation to give a meaningful result (e.g., "What is Thursday divided by red?" is absurd). However, it is possible to do an operation on mixed data (e.g., "How many red shoes did we sell on Thursday?").

11.3 MOLAP

MOLAP, or multidimensional OLAP, is the "data in a grid" version that Dr. Codd described in his paper. This is where the idea of saving summary results first appeared. Generally speaking, MOLAP does fast simpler calculations on smaller databases. Business users have gotten very good with the design of spreadsheets over the last few decades and that same technology can be used by MOLAP engines.

The spreadsheet model is the first (and often the only) declarative language that most people learn. The advantage is that there are more spreadsheet users than SQL programmers in the world. They do not have to learn a new conceptual framework, just a new language for it.

11.4 ROLAP

ROLAP, or relational OLAP, was developed after MOLAP. The main difference is that ROLAP does not do precomputation or store summary data in the database. ROLAP tools create dynamic SQL queries when the user requests the data. The exception to that description is the use of materialized `VIEW`s, which will persist summary data for queries that follow its first invocation. Some goals of ROLAP are to be scalable because it would reduce storage requirements and to be more flexible and portable because it uses SQL or an SQL-like language.

Another advantage is that the OLAP and transactional databases can be the same engine. RDBMS engines have gotten very good at handling large

amounts of data, working in parallel, and optimizing queries stored in their particular internal formats. It is a shame to lose those advantages. For example, DB2's optimizer now detects a star schema by looking for a single large table with many smaller ones joined to it. If it finds a star schema, it creates appropriate execution plans based on that assumption.

11.5 HOLAP

The problem with a pure ROLAP engine is that it is slower than MOLAP. Think about how most people actually work. A broad query is narrowed down to a more particular one. Particular tables, such as a general end-of-the-month summary, can be constructed once and shared among many users.

This lead to HOLAP, hybrid OLAP, which retains some of result tables in specialized storage or indexing so that they can be reused. The base tables dimension tables and some summary tables are in the RDBMS. This is probably the most common approach in products today.

11.6 OLAP Query Languages

SQL is the standard query language for transactional databases. Other than a few OLAP features added to SQL-99, there is no such language for analytics. The closest thing is the MDX language from Microsoft, which has become a de-facto standard by virtue of Microsoft's market domination.

This is not because MDX is a technically brilliant language, but because Microsoft makes it so much cheaper than other products. The syntax is a confusing mix of SQL and an OO dialect of some kind. Compared to full statistical packages, it is also weak.

There are statistical languages, such as SAS and IBM's SPSS, which have been around for decades. These products have a large number of options and a great deal of computational power. In fact, you really need to be an expert to fully use them. Today, they come with a GUI, which makes the coding easier. But that does not mean you do not need statistical knowledge to make the right decisions.

11.7 Aggregation Operators in SQL

When OLAP hit in IT, the SQL committee wanted to get ahead of the curve. A big fear was that vendors would create their own features with proprietary syntax and semantics. This would lead to dialects and mess up all that work

that had been done on standards. So SQL got OLAP functions.

OLAP functions add the ROLLUP and CUBE extensions to the GROUP BY clause. The ROLLUP and CUBE extensions are often referred to as supergroups. They can be written in older standard SQL using GROUP BY and UNION operators, but it is insanely complex. It is always nice to be able to define a new feature as a shorthand for older operations. The compiler writers can reuse some of the code they already have and the programmers can reuse some of the mental model they already have.

But in the case of SQL, it also means that the results of these new features will also be SQL tables and not a new kind of data structure, like the classic GROUP BY result sets.

11.7.1 GROUP BY GROUPING SET

The GROUPING SET (< column list >) is shorthand starting in SQL-99 for a series of UNION queries that are common in reports. For example, to find the total:

```
SELECT dept_name, CAST(NULL AS CHAR(10)) AS job_title, COUNT(*)
  FROM Personnel
  GROUP BY dept_name
UNION ALL
SELECT CAST(NULL AS CHAR(8)) AS dept_name, job_title, COUNT(*)
  FROM Personnel
  GROUP BY job_title;
```

which can be rewritten as:

```
SELECT dept_name, job_title, COUNT(*)
  FROM Personnel
  GROUP BY GROUPING SET (dept_name, job_title);
```

There is a problem with all of the new grouping functions. They will generate NULLs for each dimension at the subtotal levels. How do you tell the difference between a real NULL that was in the original data and a generated NULL? This is a job for the GROUPING() function, which returns 0 for NULLs in the original data and 1 for generated NULLs that indicate a subtotal.

Here is a little trick to get a human-readable output:

```
SELECT CASE GROUPING(dept_name)
```

```
   WHEN 1 THEN 'department total'
   ELSE dept_name END AS dept_name,
   CASE GROUPING(job_title)
   WHEN 1 THEN 'job total'
   ELSE job_title_name END AS job_title
   FROM Personnel
GROUP BY GROUPING SETS (dept_name, job_title);
```

This is actually a bad programming practice since display should be done in the front end and not in the database. Another problem is that you would probably want to use an `ORDER BY` on the query, rather than get the report back in a random order. But we do not care about that in SQL.

The grouping set concept can be used to define other OLAP groupings.

11.7.2 ROLLUP

A `ROLLUP` group is an extension to the `GROUP BY` clause that produces a result set that contains subtotal rows in addition to the regular grouped rows. Subtotal rows are superaggregate rows that contain further aggregates of which the values are derived by applying the same column functions that were used to obtain the grouped rows. A `ROLLUP` grouping is a series of grouping sets.

This is a "control break report" in classic COBOL and report writers used with sequential file processing:

`GROUP BY ROLLUP (a, b, c)` is equivalent to

```
GROUP BY GROUPING SETS
(a, b, c)
(a, b)
(a)
()
```

Notice that the *n* elements of the ROLLUP translate to a (*n* + 1) grouping set. Another point to remember is that the order in which the grouping expression is specified is significant for `ROLLUP`.

The `ROLLUP` is basically the classic totals and subtotals report presented as an SQL table. The following example is a simple report for three sales regions. The `ROLLUP` function is used in the `GROUP BY` clause:

```
SELECT B.region_nbr, S.city_id, SUM(S.sale_amt) AS total_sales
  FROM SalesFacts AS S, MarketLookup AS M
  WHERe S.city_id = B.city_id
  AND B.region_nbr IN (1, 2, 6)
GROUP BY ROLLUP(B.region_nbr, S.city_id)
ORDER BY B.region_nbr, S.city_id;
```

The SELECT statement behaves in the usual manner. That is, the FROM clause builds a working table, the WHERE clause removes rows that do not meet the search conditions, and the GROUP BY clause breaks the data into groups that are then reduced to a single row of aggregates and grouping columns. A sample result of the SQL is shown in Table 11.1. The result shows ROLLUP of two groupings (region, city) returning three totals, including region, city, and grand total.

Table 11.1

Yearly Sales by City and Region

Region Number	City ID	Total Sales and Comment
1	1	81
2	2	13
…	…	…
2	NULL	1123 – region #2
3	11	63
3	12	110
…	…	…
3	NULL	1212 –region #3
6	35	55
6	74	13
…	…	…
6	NULL	902 –region #6
NULL	NULL	3237 –grand total

11.7.3 CUBE

The CUBE supergroup is the other SQL-99 extension to the GROUP BY clause that produces a result set that contains all the subtotal rows of a ROLLUP aggregation and, in addition, contains cross-tabulation rows. Cross-tabulation rows are additional superaggregate rows. They are, as the name implies, summaries across columns if the data were represented as a spreadsheet. Like ROLLUP, a CUBE group can also be thought of as a series of grouping sets. In

the case of a CUBE, all permutations of the cubed grouping expression are computed along with the grand total. Therefore, the *n* elements of a CUBE translate to (2*n*) grouping sets. For example:

```
GROUP BY CUBE (a, b, c)
```

is equivalent to

```
GROUP BY GROUPING SETS
(a, b, c) (a, b) (a, c) (b, c) (a) (b) (c) ()
```

Notice that the three elements of the CUBE translate to eight grouping sets. Unlike ROLLUP, the order of specification of elements doesn't matter for CUBE: CUBE (a, b) is the same as CUBE (b, a). But the rows might not be produced in the same order, depending on your product.

Table 11.2

Sex and Race

Sex Code	Race Code	Totals and Comment
M	Asian	14
M	White	12
M	Black	10
F	Asian	16
F	White	11
F	Black	10
M	NULL	36 - Male Tot
F	NULL	37 – Female Tot
NULL	Asian	30 - Asian Tot
NULL	White	23 – White Tot
NULL	Black	20 – Black Tot
NULL	NULL	73 – Grand Tot

CUBE is an extension of the ROLLUP function. The CUBE function not only provides the column summaries we saw in ROLLUP but also calculates the row summaries and grand totals for the various dimensions. This is a version of cross-tabulations (cross-tabs) that you know from statistics. For example:

```
SELECT sex_code, race_code, COUNT(*) AS total
  FROM Census
  GROUP BY CUBE(sex_code, race:code);
```

11.7.4 Notes about Usage

If your SQL supports these features, you need to test to see what GROUPING() does with the NULLs created by an outer join. Remember that SQL does not have to return the rows in a table in any particular order. You will still have to put the results into a CURSOR with an ORDER BY clause to produce a report. But you may find that the results tend to come back in sorted order because of how the SQL engine does its work.

This has happened before. Early versions of SQL did GROUP BY operations with a hidden sort; later SQL products used parallel processing, hashing, and other methods to form the groupings that did not have a sort as a side-effect. Always write standard SQL and do not depend on the internals of one particular release of one particular SQL product.

11.8 OLAP Operators in SQL

IBM and Oracle jointly proposed extensions in early 1999 and thanks to ANSI's uncommonly rapid actions, they became part of the SQL-99 standard. IBM implemented portions of the specifications in DB2 UDB 6.2, which was commercially available in some forms as early as mid-1999. Oracle 8i version 2 and DB2 UDB 7.1, both released in late 1999, contained most of these features.

Other vendors contributed, including database tool vendors Brio, MicroStrategy, and Cognos, and database vendor Informix (not yet part of IBM), among others. A team lead by Dr. Hamid Pirahesh of IBM's Almaden Research Laboratory played a particularly important role. After his team had researched the subject for about a year and come up with an approach to extending SQL in this area, he called Oracle. The companies then learned that each had independently done some significant work. With Andy Witkowski playing a pivotal role at Oracle, the two companies hammered out a joint standards proposal in about two months. Red Brick was actually the first product to implement this functionality before the standard, but in a less complete form. You can find details in the ANSI document "Introduction to OLAP Functions" by Fred Zemke, Krishna Kulkarni, Andy Witkowski, and Bob Lyle.

11.8.1 OLAP Functionality

OLAP functions are a bit different from the GROUP BY family. You specify a "window" defined over the rows over to which an aggregate function is applied, and in what order. When used with a column function, the applicable

rows can be further refined, relative to the current row, as either a range or a number of rows preceding and following the current row. For example, within a partition by month, an average can be calculated over the previous three-month period.

Row Numbering

While SQL is based on sets that have no ordering, people depend on ordering to find data. Would you like to have a randomized phone book? In SQL-92 the only ways to add row numbering to a result were to use a cursor (in effect, making the set into a sequential file) or to use a proprietary feature. The vendor's features were all different.

One family uses a pseudocolumn attached to the table that adds an increasing integer to each row. The `IDENTITY` column used in SQL server is the most common example. The first practical consideration is that `IDENTITY` is proprietary and nonportable, so you know that you will have maintenance problems when you change releases or port your system to other products. Newbies actually think they will never port code! Perhaps they only work for companies that are failing and will be gone. Perhaps their code is so bad nobody else wants their application.

But let's look at the logical problems. First try to create a table with two columns and try to make them both `IDENTITY`. If you cannot declare more than one column to be of a certain data type, then that thing is not a data type at all, by definition. It is a property that belongs to the `PHYSICAL` table, not the `LOGICAL` data in the table.

Next, create a table with one column and make it an `IDENTITY`. Now try to insert, update, and delete different numbers from it. If you cannot insert, update, and delete rows from a table, then it is not a table by definition.

Finally, the ordering used is unpredictable when you insert with a `SELECT` statement, as follows:

```
INSERT INTO Foobar (a, b, c)
SELECT x, y, z
  FROM Floob;
```

Since a query result is a table, and a table is a set that has no ordering, what should the `IDENTITY` numbers be? The entire, whole, completed set is presented to `Foobar` all at once, not a row at a time. There are (*n*!) ways to number *n* rows, so which one do you pick? The answer has been to use

whatever the physical order of the result set happened to be. That nonrelational phrase “physical order” again!

But it is actually worse than that. If the same query is executed again, but with new statistics or after an index has been dropped or added, the new execution plan could bring the result set back in a different physical order. Indexes and statistics are not part of the logical model.

The second family is to expose the physical location on the disk in an encoded format that can be used to directly move the read/writer head to the record. This is the Oracle ROWID. If the disk is defragmented, the location can be changed, and the code will not port. This approach is dependent on hardware.

The third family is a function. This was originally done in Sybase SQL Anywhere (see WATCOM SQL) and was the model for the standard SQL `ROW_NUMBER()` function.

This function computes the sequential row number of the row within the window defined by an ordering clause (if one is specified), starting with 1 for the first row and continuing sequentially to the last row in the window. If an ordering clause, `ORDER BY`, isn’t specified in the window, the row numbers are assigned to the rows in arbitrary order as returned by the subselect. In actual code, the numbering functions are used for display purposes rather than adding line numbers in the front end.

A cute trick for the median is to use two `ROW_NUMBER()` with an `OVER()` clause:

```
SELECT AVG(x),
  ROW_NUMBER() OVER(ORDER BY x ASC) AS hi,
  ROW_NUMBER() OVER(ORDER BY x DESC) AS lo
  FROM Foobar
  WHERE hi IN (lo, lo + 1, lo-1);
```

This handles both the even and old number of cases. If there are an odd number of rows then (`hi = lo`). If there is an even number of rows, then we want the two values in the two rows to either side of the middle. I leave it to you to play with duplicate values in column `x` and getting a weighted median, which is a better measure of central tendency. For example:

```
x hi lo
==========
```

```
1 1 7
1 2 6
2 3 5
3 4 4 < = median – 4.0
3 5 3
3 6 2
3 7 1
```

The median for an even number of cases:

```
x hi lo
===========
1 1 6
1 2 5
2 3 4 < = median
3 4 3 < = median = 3.5
3 5 2
3 6 1
```

RANK and DENSE_RANK

So far, we have talked about extending the usual SQL aggregate functions. There are special functions that can be used with the window construct.

RANK assigns a sequential rank of a row within a window. The RANK of a row is defined as one plus the number of rows that strictly precede the row. Rows that are not distinct within the ordering of the window are assigned equal ranks. If two or more rows are not distinct with respect to the ordering, then there will be one or more gaps in the sequential rank numbering. That is, the results of RANK may have gaps in the numbers resulting from duplicate values. For example:

```
x RANK
========
1 1
2 3
2 3
3 5
3 5
```

```
3 5
3 5
3 5
3 5
```

`DENSE_RANK` also assigns a sequential rank to a row in a window. However, a row's `DENSE_RANK` is one plus the number of rows preceding it that are distinct with respect to the ordering. Therefore, there will be no gaps in the sequential rank numbering, with ties being assigned the same rank. `RANK` and `DENSE_RANK` require an `ORDER BY` clause. For example:

```
x DENSE_RANK
============
1 1
2 2
2 2
3 3
3 3
3 3
3 3
3 3
```

Aside from these functions, the ability to define a window is equally important to the OLAP functionality of SQL. You use windows to define a set of rows over which a function is applied and the sequence in which it occurs. Another way to view the concept of a window is to equate it with the concept of a slice. In other words, a window is simply a slice of the overall data domain.

Moreover, when you use an OLAP function with a column function, such as `AVG()`, `SUM()`, `MIN()`, or `MAX()`, the target rows can be further refined, relative to the current row, as either a range or a number of rows preceding and following the current row. The point is that you can call upon the entire SQL vocabulary to be combined in any of your OLAP-centric SQL statements.

Window Clause

The window clause has three subclauses: partitioning, ordering, and aggregation grouping. The general format is

```
< aggregate function > OVER ([PARTITION BY < column list >]
  ORDER BY < sort column list > [< aggregation grouping >])
```

A set of column names specifies the partitioning, which is applied to the rows that the preceding `FROM`, `WHERE`, `GROUP BY`, and `HAVING` clauses produced. If no partitioning is specified, the entire set of rows composes a single partition and the aggregate function applies to the whole set each time. Though the partitioning looks like a `GROUP BY`, it is not the same thing. A `GROUP BY` collapses the rows in a partition into a single row. The partitioning within a window, though, simply organizes the rows into groups without collapsing them.

The ordering within the window clause is like the `ORDER BY` clause in a `CURSOR`. It includes a list of sort keys and indicates whether they should be sorted ascending or descending. The important thing to understand is that ordering is applied only within each partition.

The `< aggregation grouping >` defines a set of rows upon which the aggregate function operates for each row in the partition. Thus, in our example, for each month, you specify the set including it and the two preceding rows. Here is an example from an ANSI paper on the SQL-99 features:

```
SELECT SH.region, SH.month, SH.sales,
  AVG(SH.sales)
  OVER (PARTITION BY SH.region
  ORDER BY SH.month ASC
  ROWS 2 PRECEDING)
  AS moving_average
  FROM SalesHistory AS SH;
```

Here, `AVG(SH.sales) OVER (PARTITION BY. . .)` is an OLAP function. The construct inside the OVER() clause defines the “window” of data to which the aggregate function, `AVG()` in this example, is applied.

The window clause defines a partitioned set of rows to which the aggregate function is applied. The window clause says to take the `SalesHistory` table and then apply the following operations to it:

- Partition `SalesHistory` by region.
- Order the data by month within each region.

◆ Group each row with the two preceding rows in the same region.

◆ Compute the moving average on each grouping.

The database engine is not required to perform the steps in the order described here, but has to produce the same result set as if they had been carried out.

There are two main types of aggregation groups: physical and logical. In physical grouping, you count a specified number of rows that are before or after the current row. The `SalesHistory` example uses physical grouping. In logical grouping, you include all the data in a certain interval, defined in terms of a subset positioned relative to the current sort key. For instance, you create the same group whether you define it as the current month's row plus:

1. The two preceding rows as defined by the `ORDER BY` clause.

2. Any row containing a month no less than two months earlier.

Physical grouping works well for contiguous data and programmers who think in terms of sequential files. Physical grouping works for a larger variety of data types than logical grouping, because it does not require operations on values.

Logical grouping works better for data that has gaps or irregularities in the ordering and for programmers who think in SQL predicates. Logical grouping works only if you can do arithmetic on the values, such as numeric quantities and dates.

A physical grouping is based on aggregating a fixed number of rows in a partition, based on their position relative to the row for which the function is computed. One general format is:

```
OVER (RANGE BETWEEN  < bound_1 > AND  < bound_2 >)
```

The start of the window, `< bound_1 >`, can be

```
UNBOUNDED PRECEDING

< unsigned constant > PRECEDING

< unsigned constant > FOLLOWING

CURRENT ROW
```

The meanings are obvious. Unbounded proceedings include the entire partition that precedes the current row in the sort order. The numbered displacements are done by counting rows.

The end of the window, `< bound_2 >`, can be

```
UNBOUNDED FOLLOWING
< unsigned constant > PRECEDING
< unsigned constant > FOLLOWING
CURRENT ROW
```

The `UNBOUNDED FOLLOWING` option includes the entire partition that follows the current row in the sort order. For example, you can include the whole partition:

```
OVER (RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)
```

The `ROWS` option is shorthand that involves only preceding rows. For example, this is a running accumulative total:

```
SELECT SUM(x)
  OVER (ROWS UNBOUNDED PRECEDING) AS running_total
  FROM Foobar;
```

11.8.2 NTILE(n)

`NTILE(n)` splits a set into equal groups of approximately *n* rows. This often has vendor extensions and rules about the buckets, so use it with caution. For example:

```
NTILE(3) OVER (ORDER BY x)
x NTILE
================
1 1
1 1
2 1
2 1
3 2
3 2
3 2
3 3
3 3
3 3
```

The SQL engine attempts to get the groups the same size, but this is not always possible. The goal is then to have them differ by just one row. `NTILE(n)`, where `(n)` is greater than the number of rows in the query, is effectively a `ROW_NUMBER()` with groups of size one.

Obviously, if you use `NTILE(100)`, you will get percentiles, but you need at least 100 rows in the result set. A trick to prune off outliers (a value that is outside the range of the other data values) is to use `NTILE(200)` and drop the first and 200th bucket to rule out the 0.5% on either end of the normal distribution.

11.8.3 Nesting OLAP Functions

One point will confuse older SQL programmers. These OLAP extensions are scalar functions, not aggregates. You cannot nest aggregates in standard SQL because it would make no sense. Consider this example:

```
SELECT customer_id, SUM(SUM(purchase_amt)) –error
  FROM Sales
  GROUP BY customer_id;
```

Each customer should get a total of his or her purchases with the innermost `SUM()`, which is one number for the grouping. If it worked, the outermost `SUM()` would be the total of that single number. However, you can write

```
SUM(SUM(purchase_amt)OVER (PARTITION BY depart_nbr))
```

In this case the total purchase amount for each department is computed, and then summed.

11.8.4 Sample Queries

Probably the most common use of row numbering is for display in the front end. This is not a good thing, since display is supposed to be done in the front end and not in the database. But here it is:

```
SELECT invoice_nbr,
  ROW_NUMBER()
  OVER (ORDER BY invoice_nbr) AS line_nbr,
  FROM Invoices
  ORDER BY invoice_nbr;
```

Now let’s try something that is more like a report. List the top five wage

earners in the company:

```
SELECT emp_nbr, last_name, sal_tot, sal_rank
  FROM (SELECT emp_nbr, last_name, (salary + bonus)
  RANK()
  OVER (ORDER BY (salary + bonus) DESC)
  FROM Personnel)
  AS X(emp_nbr, last_name, sal_tot, sal_rank)
  WHERE sal_rank < 6;
```

The derived table `X` computes the ranking, and then the containing query trims off the top five.

Given a table of sales leads and dealers, we want to match them based on their ZIP codes. Each dealer has a priority and each lead has a date on which it was received. The dealers with the highest priority get the earlier leads:

```
(SELECT lead_id,
  ROW_NUMBER()
  OVER (PARTITION BY zip_code
  ORDER BY lead_date)
  AS lead_link
  FROM Leads) AS L
FULL OUTER JOIN
(SELECT dealer_id,
  ROW_NUMBER()
  OVER (PARTITION BY zip_code
  ORDER BY dealer_priority DESC)
  AS dealer_link
  FROM Dealers) AS D
ON D.dealer_link = L.lead_link
AND D.zip_code = L.zip_code;
```

You can add more criteria to the `ORDER BY` list, or create a lookup table with multiparameter scoring criteria.

11.9 Sparseness in Cubes

The concept of cubes is a nice way to get a graphic image when you are thinking about reporting data. Think about a simple spreadsheet with columns that hold the dates of a shipment and rows that hold the particular product shipped on that date. Obviously, this is going to be a large spreadsheet if the company has, say, 10,000 items and five years of history to look at (3,652,500 cells, actually).

Most of these cells will be empty. But there is a subtle difference between empty, zero, and NULL. Empty is a spreadsheet term and it means the cell exists because it was created by the range of rows and columns when the spreadsheet was set up. Zero is a numeric value; it needs to be in a cell to exist and you cannot divide by it—it is really a number. NULLs are an SQL concept that holds a place for a missing or unknown value. Remember that NULLs propagate in simple computations and they require storage in SQL; they are not the same as an empty cell.

Imagine that we did not ship iPods before October 23, 2001 because they were not released before that date. This cell does not exist in the cube yet. And in October 2001, we had an inventory of red-velvet, hip-hugger bell-bottoms; we sold none of them (the same as every month since 1976); this is a zero. Finally, nobody has reported iPod shipments for March 25, 2006 yet, but we know it is a hot item and we will have made sales. Now we add a column for iPods and a history of empty cells appear.

At this point, you need to decide how to handle these cases. I would recommend ignoring nonexistent iPods in any sales history reports. Your cube tool should be able to tell the difference between all three cases. But the long history of not selling red-velvet, hip-hugger bell-bottoms (i.e., shipments = 0) is important information—hey, disco is dead and you need to clean out the inventory.

A NULL is also information, but more properly a sign that data is missing. This is trickier because you need to have methods for handling that missing data.

Can you estimate a value and use it? If my iPod sales have increased at a steady rate of *p*% per month for the last *m* months, can I assume the trend will continue? Or should I use a median or an average? Or should I use data up to the point that I know values?

11.9.1 Hypercube

Extend the spreadsheet model from two dimensions to three, four, five, and so

forth. Human beings have serious problems with a graphic greater than three dimensions. The universe in which we live and see things is limited to three dimensions.

Enough of the cells will be empty that it is vital that the storage for the cube have a sparse matrix implementation. That means we do not physically store empty cells, but they might be materialized inside the engine. Figure 11.1 is a diagram for a (sources, routes, time) cube. I will explain the hierarchies on the three axes.

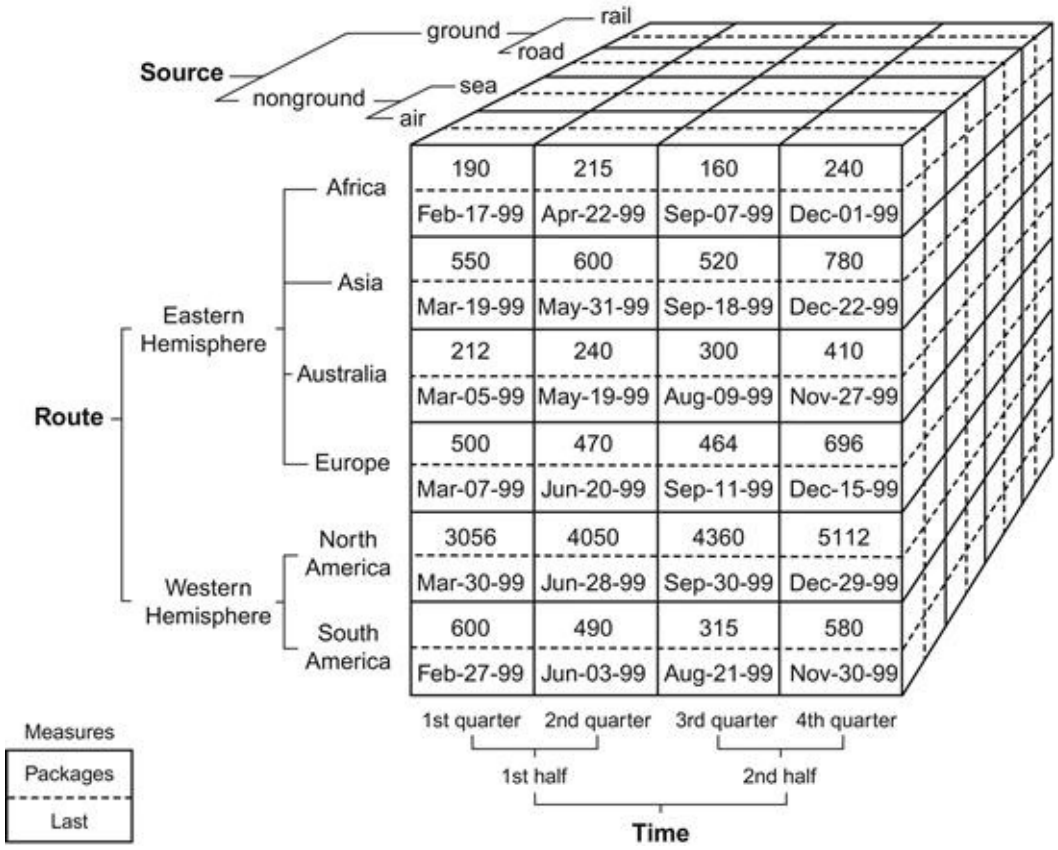


FIGURE 11.1 A (sources, routes, time) cube. Source: *http://www.aspfree.com/c/a/MS-SQL-Server/Accessing-OLAP-using-ASP-dot-NET/*

11.9.2 Dimensional Hierarchies

A listing of all of the cells in a hypercube is useless. We want to see aggregated (summary) information. We want to know that iPod sales are increasing, that we need to get rid of those bell-bottoms, that most of our business is in North America, and so forth.

Aggregation means that we need a hierarchy on the dimensions. Here is an example of a temporal dimension. Writing a hierarchy in SQL is easily done with a nested sets model if you have to write your own code:

```
CREATE TABLE TemporalDim
(range_name CHAR(15) NOT NULL PRIMARY KEY,
range_start_date DATE NOT NULL,
range_end_date DATE NOT NULL,
  CHECK (range_start_date < range_end_date));
INSERT INTO TemporalDim
VALUES ('Year2006', '2006-01-01', '2006-12-31'),
  ('Qtr-01-2006', '2006-01-01', '2006-03-31'),
  ('Mon-01-2006', '2006-01-01', '2006-01-31'),
  ('Day:2006-01-01', '2006-01-01', '2006-01-01'),
  ..;
```

You can argue that you do not need to go to the leaf node level in the hierarchy, but only to the lowest level of aggregation. That will save space, but the code can be trickier when you have to show the leaf node level of the hierarchy.

This is the template for a hierarchical aggregation:

```
SELECT TD.range_name, ..
  FROM TemporalDim AS TD, FactTable AS F
  WHERE F.shipping_time BETWEEN TD.range_start_date
  AND TD.range_end_date;
```

If you do not have the leaf nodes in the temporal dimension, then you need to add a CTE (common table expression) with the days and their names that are at the leaf nodes in your query:

```
WITH Calendar (date_name, cal_date)
AS VALUES (CAST ('2006-01-01' AS CHAR(15)),
  CAST ('2006-01-01' AS DATE),
  ('2006-01-02', '2006-01-02'),
  etc.
SELECT TD.range_name, ..
  FROM Temporal_Dim AS TD, Fact_Table AS F
  WHERE F.shipping_time BETWEEN TD.range_start_date
  AND TD.range_end_date
UNION ALL
```

```
SELECT Calendar.date_name, ..
  FROM Calendar AS C, FactTable AS F
  WHERE C.cal_date = F.shipping_time;
```

A calendar table is used for other queries in the OLTP side of the house, so you should already have it in at least one database. You may also find that your cube tool automatically returns data at the leaf level if you ask it.

11.9.3 Drilling and Slicing

Many of the OLAP user interface tools will have a display with a dropdown menu for each dimension that lets you pick the level of aggregation in the report. The analogy is the slide switches on the front of expensive stereo equipment, which set how the music will be played, but not what songs are on the CD. It is called drilldown because you start at the highest level of the hierarchies and travel down the tree structure. Figure 11.2 is an example of such an interface from a Microsoft reporting tool.

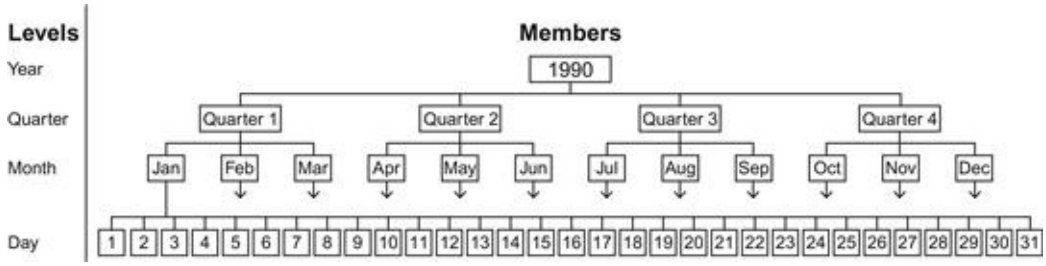


FIGURE 11.2 Example of a drilldown tree structure. Source: *http://www.aspfree.com/c/a/MS-SQL-Server/Accessing-OLAP-using-ASP-dot-NET/*

Slicing a cube is another physical analogy. Look at the illustration of the cube in Figure 11.1 and imagine that you have a cheese knife and you slice off blocks from the cube. For example, you might want to look at only the ground shipments, so you slice off just that dimension, building a subcube. The drilldowns will still be in place. This is like picking the song from a CD.

Concluding Thoughts

We invented a new occupation: data scientist! This does not have an official definition, but it seems to be a person who can use OLAP databases, statistics packages, and some of the NoSQL tools we have discussed. He or she also seems to need a degree in statistics or math. There have also been a lot of advances in statistics that use the improved, cheap computing power we have today. For example, the assumption of a normal distribution in data was made for convenience of computations for centuries. Today, we can use resampling,

pareto, and exponential models, and even more complex math.

References

1. Codd EF. *Providing OLAP to User-Analysts: An IT Mandate*. Available at, *http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf*; 1993.
2. Zemke F, Kulkarni K, Witkowski A, Lyle B. *Introduction to OLAP functions (ISO/IEC JTC1/ SC32 WG3:YGJ-068 ANSI NCITS H2-99-154r2)*. 1999; In: *ftp://ftp.iks-jena.de/mitarb/lutz/standards/sql/OLAP-99-154r2.pdf*; 1999.

CHAPTER 12

Multivalued or NFNF Databases

Abstract

Multivalued or NFNF databases violate 1NF by allowing a column to contain more than one scalar value. This model has been use for decades in commercial products but did not get a theoretical basis until much later. This chapter discusses multivalued databases.

Keywords

COBOL; Pick; nonfirst normal form (NFNF); first normal form (1NF)

Introduction

RDBMS is based on first normal form (1NF), which assumes that data is kept in scalar values in columns that are kept in rows and those records have the same structure. The multivalued model allows tables to be nested inside columns. They have a niche market that is not well known to SQL programmers. There is an algebra for this data model that is just as sound as the relational model.

In 2013, most current database programmers have never worked with anything but SQL. They did not grow up with file systems, COBOL, or any of the old network or hierarchical databases. These products are still around and run a lot of commercial applications. But we also have seen a blending of the traditional hierarchical sequential records and the set-oriented models of data. Let’s start with some history.

12.1 Nested File Structures

A flat file is a file of which the fields are all scalar values and the records have the same structure. When the relational model first came out in the 1970s, developers mistook tables for flat files. They missed the mathematics and the idea of a set. A file is read from left to right, in sequence, record by record; a table exists as a set that is the unit of work. A file stands alone while a table is part of a schema and has relationships with the rest of the schema. Frankly, a

lot of developers still do not understand these concepts.

But a flat file is the easiest starting point. The next most complicated file structures have variant records. Since a file is read left to right, it can tell the computer what to expect “downstream” in the data. In COBOL, we use the `OCCURS` and `OCCURS DEPENDING ON` clauses.

I will assume most of the readers do not know COBOL. The superquick explanation is that the COBOL `DATA DIVISION` is like the DDL in SQL, but the data is kept in strings that have a picture (`PIC`) clause that shows their display format. In COBOL, display and storage formats are the same. Records are made of a hierarchy of fields, and the nesting level is shown as an integer at the start of each declaration (numbers increase with depth; the convention is to step by fives). Suppose you wanted to store your monthly sales figures for the year. You could define 12 fields, one for each month, like this:

```
05 MONTHLY-SALES-1 PIC S9(5)V99.
05 MONTHLY-SALES-2 PIC S9(5)V99.
05 MONTHLY-SALES-3 PIC S9(5)V99.
. . .
05 MONTHLY-SALES-12 PIC S9(5)V99.
```

The dash is like an SQL underscore, a period is like a semicolon in SQL, and the picture tells us that each sales amount has a sign, up to five digits for dollars and two digits for cents. You can specify the field once and declare that it repeats 12 times with the simple `OCCURS` clause, like this:

```
05 MONTHLY-SALES OCCURS 12 TIMES PIC S9(5)V99.
```

The individual fields are referenced in COBOL by using subscripts, such as `MONTHLY-SALES(1)`. The `OCCURS` can also be at the group level, and this is its most useful application. For example, all 25 line items on an invoice (75 fields) could be held in this group:

```
05 LINE-ITEMS OCCURS 25 TIMES.
10 ITEM-QUANTITY PIC 9999.
10 ITEM-DESCRIPTION PIC X(30).
10 UNIT-PRICE PIC S9(5)V99.
```

Notice the `OCCURS` is listed at the group level, so the entire group occurs 25 times.

There can be nested `OCCURS`. Suppose we stock 10 products and we want to keep a record of the monthly sales of each product for the past 12 months:

```
01 INVENTORY-RECORD.
  05 INVENTORY-ITEM OCCURS 10 TIMES.
  10 MONTHLY-SALES OCCURS 12 TIMES PIC 999.
```

In this case, `INVENTORY-ITEM` is a group composed only of `MONTHLY-SALES`, which occurs 12 times for each occurrence of an inventory item. This gives an array of 10 × 12 fields. The only information in this record is the 120 monthly sales figures—12 months for each of 10 items.

Notice that `OCCURS` defines an array of known size. But because COBOL is a file system language, it reads fields in records from left to right. Since there is no `NULL`, inserting future values that are not yet known requires some coding tricks. The language has the `OCCURS DEPENDING ON` option. The computer reads an integer control field and then expects to find that many occurrences of a subrecord following at runtime. Yes, this can get messy and complicated, but look at this simple patient medical treatment history record to get an idea of the possibilities:

```
01 PATIENT-TREATMENTS.
05 PATIENT-NAME PIC X(30).
05 PATIENT-NUMBER PIC 9(9).
05 TREATMENT-COUNT PIC 99 COMP-3.
05 TREATMENT-HISTORY OCCURS 0 TO 50 TIMES
DEPENDING ON TREATMENT-COUNT
INDEXED BY TREATMENT-POINTER.
10 TREATMENT-DATE.
15 TREATMENT-YEAR PIC 9999.
15 TREATMENT-MONTH PIC 99.
15 TREATMENT-DAY PIC 99.
10 TREATING-PHYSICIAN-NAME PIC X(30).
10 TREATMENT-CODE PIC 999.
```

The `TREATMENT-COUNT` has to be handled in the applications to correctly describe the `TREATMENT-HISTORY` subrecords. I will not explain `COMP-3` (a data type for computations) or the `INDEXED BY` clause (array index), since they are not important to my point.

My point is that we had been thinking of data in arrays of nested structures before the relational model. We just had not separated data from computations and presentation layers, nor were we looking for an abstract model of

computing yet.

12.2 Multivalued Systems

When mini-computers appeared, they had limited capacity compared to current hardware and software. Products that were application development systems that integrated a database with an application language were favorites among users. You could build a full application with one tool!

One of the most successful such tools was Pick. This began life as the Generalized Information Retrieval Language System (GIRLS) on an IBM System/360 in 1965 by Don Nelson and Dick Pick at TRW for use by the U.S. Army to control the inventory of Cheyenne helicopter parts.

The relational model did not exist at this time; in fact, we really did not have any data theory. The Pick file structure was made up of variable-length strings. This is not the model used in COBOL. In Pick, records are called items, fields are called attributes, and subfields are called values or subvalues (hence the present-day term *multivalued database*). All elements are variable length, with field and values marked off by special delimiters, so that any file, record, or field may contain any number of entries of the lower level of entity.

As a result, a Pick item (record) can be one complete entity (e.g., an entire, complete customer order) rather than an RDBMS model with a table of the set of all order headers that relates to the set of all customer order details, which relates to the inventory, etc.

The Pick system is written for a virtual machine and it included Unix-like hierarchy of directories, subdirectories, and files with records being hashed into buckets that could be scanned. There is a data dictionary that holds the system together. It also comes with a command-line language, so it is self-contained.

This made porting Pick to other platforms easy. It was quickly licensed by many distributors, so Pick became a generic name for the family of multivalued databases with an implementation of Pick/BASIC. Dick Pick founded Pick & Associates, later renamed Pick Systems, then Raining Data, and, as of 2011, TigerLogic. He licensed Pick to a large variety of manufacturers and vendors who have produced different dialects of Pick. The dialect sold by TigerLogic is now known as D3, mvBase, and mvEnterprise. Those previously sold by IBM under the U2 umbrella are known as UniData and UniVerse. Rocket Software purchased IBM’s U2 line in 2010.

Pick runs on a large assortment of microcomputers, personal computers, and mainframe computers, and is still in use as of 2013. Here is a short list of Pick family products:

- UniVerse (Unix based)
- UniData (Unix based)
- D3
- jBASE
- ARev
- Advanced Pick
- mvBase
- mvEnterprise
- R83

If you are old enough to remember dBase from Ashton-Tate as the first popular database product on a PC, you can compare this to how that line of development became the Xbase family of products.

Pick was first released commercially in 1973 by Microdata Corporation (and their British distributor CMC) as the Reality Operating System now supplied by Northgate Information Solutions. The Microdata implementation added a BASIC-like language called Pick/BASIC (see Data/BASIC). This became the de-facto Pick development language because it has extensions for smart terminal interface as well as the database operations.

Eventually, like all NoSQL products seem to do, they added a SQL-style language called ENGLISH (later, ACCESS, not be confused with the Microsoft database system of the same name) for retrieval and reporting. ENGLISH could not do updates at first, but later added the command `REFORMAT` for batch updating. ENGLISH did not have joins or other relational operators. In effect, you “prejoined” tables in Pick by using the data dictionary redefinitions for a field, which would execute a calculated lookup in another file. Data integrity has to be done in the application code.

Proprietary variations and enhancements sprouted up, but the core product has remained the same. Pick is used primarily for business data processing because its data model matches neatly to files and forms used in office work and it has a strong enthusiastic user community.

12.3 NFNF Databases

Programming languages have had a formal basis, such as FORTRAN being based on Algebra, LISP on list processing, and so forth. Data and databases did not get “academic legitimacy” until Dr. Codd invented his relational algebra. It had everything academics love—a set of math symbols, including some new ones that would drive the typesetters crazy. But it also had axioms, thanks to Armstrong.

The immediate result was a burst of papers using Dr. Codd’s relational algebra. But the next step for a modern academic is to change or drop one of the axioms to see tha you can still have a consistent formal system. In geometry, change the parallel axiom (parallel lines never meet) to something else. For example, the replacement axiom is that two parallel lines (great circles) meet at two points on the surface of a sphere. Spheres are real. and we could test the new geometry with a real-world model.

Since 1NF is the basis for RDBMS, it was the one academics played with first. And we happen to have real multivalued databases to see if it works. Most of the academic work was done by Jaeschke and Schek at IBM and Roth, Korth, and Silberschatz at the University of Texas, Austin. They added new operators to the relational algebra and calculus to handle “nested relations” while still keeping the abstract set-oriented nature of the relational model. 1NF is not convenient for handling data with complex internal structures, such as computer-aided design and manufacturing (CAD/CAM). These applications have to handle structured entities while the 1NF table only allows atomic values for attributes.

Nonfirst normal form (NFNF) databases allow a column in a table to hold nested relations, and break the rule about a column only containing scalar values drawn from a known domain. In addition to NFNF, these databases are also called 2NF, NF^2, and ¬NF in the literature. Since they are not part of the ANSI/ISO standards, you will find different proprietary implementations and academic notations for their operations.

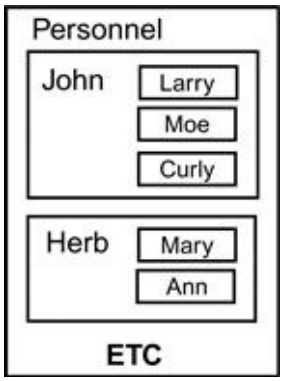
Consider a simple example of employees and their children. On a normalized schema, the employees would be in one table and their children would be in a second table that references the parent:

```
CREATE TABLE Personnel
(emp_name VARCHAR(20) NOT NULL PRIMARY KEY,
  ..);
CREATE TABLE Dependents
```

```
(dependent_name VARCHAR(20) NOT NULL PRIMARY KEY,
  emp_name VARCHAR(20) NOT NULL
  REFERENCES Personnel(emp_name)– DRI actions
  ON UPDATE CASCADE
  ON DELETE CASCADE,
..);
```

But in an NFNF schema, the dependents would be in a column with a table type, perhaps something like this:

```
CREATE NF TABLE Personnel
(emp_name VARCHAR(20) NOT NULL PRIMARY KEY,
  dependents TABLE
  (dependent_name VARCHAR(20) NOT NULL PRIMARY KEY,
  emp_name VARCHAR(20) NOT NULL,
  ..),
  ..);
```



We can extend the basic set operators `UNION`, `INTERSECTION`, and `DIFFERENCE` and subsets in a natural way. Extending the relational operations is also not difficult for `PROJECTION` and `SELECTION`. The `JOIN` operators are a bit harder, but if you restrict your algebra to the natural or equijoin, life is easier. The important characteristic is that when these extended relational operations are used on flat tables, they behave like the original relational operations.

To transform this NFNF table back into a 1NF schema, you would use an `UNNEST` operator. The unnesting, in this case, would make `Dependents` into its own table and remove it from `Personnel`. Although `UNNEST` is the mathematical inverse to `NEST`, the operator `NEST` is not always the mathematical inverse of `UNNEST` operations. Let’s start with a simple, abstract

nested table:

G1			
F1	F2	G2	
		F3	F4
X	Y	X	Y
		Y	X
X	Y	Y	Z
		Z	X

The `UNNEST(< subtable >)` operation will “flatten” a subtable up one level in the nesting:

G1			
F1	F2	F3	F4
X	Y	X	Y
X	Y	Y	X
X	Y	Y	Z
X	Y	Z	X

The nest operation requires a new table name and its columns as a parameter. This is the extended SQL declaration:

```
NEST (G1, G2(F3, F4))
```

G1			
F1	F2	G2	
		F3	F4
X	Y	X	Y
		Y	X
		Y	Z
		Z	X

When we try to “re-nest” this step back to the original table, it fails.

There is also the question of how to order the nesting. We put the dependents inside the personnel in the first `Personnel` example. Children are weak entities; they have to have a parent (a strong entity) to exist. But we could have nested the parents inside the dependents. The problem is that `NEST()` does not commute. An operation is commutative when (A ○ B) = (B ○ A), if you forgot your high school algebra. Let’s start with a simple flat table:

G1			
F1	F2	F3	F4
X	Y	X	X
X	Y	X	Y
X	Y	X	Z
X	Y	Y	X
X	Y	Z	X

Now, we do two nestlings to create subtables G2 made up of the `F3` column and `G3` made up of the `F4` column. First in this order:

```
NEST(NEST (G1, G2(F3)) , G3(F4))
```

G1			
F1	F2	G2	G3
		F3	F4
X	Y	X	X
		Y	
		Z	
X	Y	X	Y
			Z

Now in the opposite order:

```
NEST(NEST (G1, G3(F3)) , G2(F4))
```

G1			
F1	F2	G2	G3
		F3	F4
X	Y	X	X
			Y
			Z
X	Y	Y	X
		Z	

The next question is how to handle missing data. What if Herb’s daughter Mary is lactose-intolerant and has no favorite ice cream flavor in the `Personnel` table example? The usual NFNF model will require explicit markers instead of a generic missing value.

Another constraint required is for the operators to be objective, which is covered by the partitioned normal form (PNF). This normal form cannot have empty subtables and operations have to be reversible. A little more formally, a relation in PNF is such that its atomic attributes are a superkey of the relation and that any nonatomic component of a tuple of the relation is also in PNF.

12.4 Existing Table-Valued Extensions

Existing SQL products have added some NFNF extensions, but they are not well optimized. The syntax is usually dialect, even though there are some ANSI/ISO standards for them.

12.4.1 Microsoft SQL Server

Microsoft SQL Server 2008 added table-valued parameters and user-defined data types. The syntax lets you declare a table as a type, then use that type name to define a local variable or parameter in stored procedures. According to Microsoft, a table-valued parameter (TVP) is an efficient option for up to 1,000 or so rows. The syntax is straightforward:

```
CREATE TYPE Payments
AS TABLE (account_nbr CHAR(5) NOT NULL,
  payment_amt DECIMAL(12,2));
```

This is not actually an NFNF implementation; it is more of a shorthand with limitations. You cannot use user-defined data types in a base table declaration! The T-SQL dialect uses a prefixed `@` for local variables, both scalars and table variables, so to get a local, temporary table in the session, you have to write:

```
DECLARE @atm_money Payments;
```

12.4.2 Oracle Extensions

Oracle has a more powerful implementation than T-SQL, but it cannot handle optimizations without flattening the nested tables. The syntax uses Oracle's object type in the DDL; there are other collection types in the product, but let's create an `Address_Type` for a simple example. This DDL will give us an `Address_Type` and `Address_Book` table. The `Address_Book` is a table of `Address_Type`, pretty much the same syntax model at we just saw in T-SQL dialect:

```
CREATE TYPE Address_Type AS OBJECT
(addr_line VARCHAR2(35),
  city_name VARCHAR2(25),
  state_code CHAR2(2),
  zip_code CHAR2(5));
```

```
CREATE TYPE Address_Book
AS TABLE OF Address_Type;
```

So now, to create a table, we just need to specify a column name (`emp_addresses` in this case) and our newly created type (`Address_Book`):

```
CREATE TABLE Personnel
(emp_name VARCHAR2(25),
  emp_addresses Address_Book);
```

To use this table and nested table we first put the addresses objects into a base table, say `Personnel`. The nested table is now empty, so we have to fill it with `INSERT INTO` statements using the key column name. To generate a result set we can make use of the table function:

```
INSERT INTO Personnel(name, emp_addresses)
  VALUES ('Fred Flintstone', Address_Book());
INSERT INTO TABLE
(SELECT emp_addresses
  FROM Personnel
  WHERE emp_name = 'Fred Flintstone')
  VALUES ('123 Main', 'Bedrock', 'TX', '78787');
INSERT INTO TABLE
(SELECT emp_addresses
  FROM Personnel
  WHERE emp_name = 'Fred Flintstone')
  VALUES ('12 Lava Ln', 'Slag Town', 'CA', '98989');
INSERT INTO TABLE
(SELECT emp_addresses
  FROM Personnel
  WHERE emp_name = 'Fred Flintstone')
  VALUES ('77 Cave Ct', 'Pre-York', 'NY', '12121');
```

To see the results, we can flatten the nesting with a simple query. The `emp_addresses` table is treated something like a derived table expression, but there is an implied join condition:

```
SELECT T1.emp_name, T2.*
FROM Personnel AS T1,
```

```
TABLE(T1.emp_addresses) AS T2;
```

Fred Flintstone	123 Main St	Bedrock	TX	78787
Fred Flintstone	12 Lava Ln	Slag Town	CA	98989
Fred Flintstone	77 Cave Ct	Pre-York	NY	33333

Concluding Thoughts

Multivalue databases is probably a better name than nonfirst normal form databases for this family of products. It is better to define something by what it is by its nature and not as the negative of something else. Most database programmers have classes on the relational model, but nobody outside a niche has any awareness of any other models, such as the multivalue one discussed here.

CHAPTER 13

Hierarchical and Network Database Systems

Abstract

The first true databases were based on hierarchical and network database models. These products still dominate commercial data today on mainframe commercial platforms. Failure to understand them means that you will never be able to move out of legacy systems to other platforms. This database model was defined by Charles Bachman and it influenced many other directions.

Keywords

ANSI X3H2 Database Standards Committee; Charles Bachman; CDL; CODASYL; DL/I; hierarchical database systems; IDMS; IDS (Integrated Data Store); IMS; NDL (Network Database Language); network database systems

Introduction

IMS and IDMS are the most important prerelational technologies that are still in wide use today. In fact, there is a good chance that IMS databases still hold more commercial data than SQL databases. These products still do the “heavy lifting” in banking, insurance, and large commercial applications on mainframe computers and they use COBOL. They are great for situations that do not change much and need to move around a lot of data. Because so much data still lives in them, you have to at least know the basics of hierarchical and network database systems to get to the data to put it in a NoSQL tool.

I am going to assume that most of the readers of this book have only worked with SQL. If you have heard of a hierarchical or network database system, it was probably mentioned in a database course in college and then forgotten. In some ways, this is too bad. It helps to know how the earlier tools worked, so you can see how the new tools evolve from the old ones.

13.1 Types of Databases

The classic types of database structures are network, relational, and hierarchical. The relational model is associated with Dr. E. F. Codd, and the other two models are associated with Charles Bachman, who did pioneering in the 1950s at Dow Chemical, and in the 1960's at General Electric, where he developed the Integrated Data Store (IDS), one of the first database management systems. The network and hierarchical models are called "*navigational*" databases because the mental model of data access is that of a reader moving along paths to pick up the data. In fact, when Bachman received the ACM Turing Award in 1973 for his outstanding contributions to database technology, this is how he described it.

IMS was not the only navigational database, just the most popular. TOTAL from Cincom was based on a master record that had pointer chains to one or more sets of slave records. Later, IDMS and other products generalized this navigational model.

CODASYL, the committee that defined COBOL, came up with a standard for the navigational model. Finally, the ANSI X3H2 Database Standards Committee took the CODASYL model, formalized it a bit, and produced the NDL language specification. However, at that point, SQL had become the main work of the ANSI X3H2 Database Standards Committee and nobody really cared about NDL and the standard simply expired.

IMS from IBM is the most popular hierarchical database management system still in wide use today. It is stable, well defined, scalable, and very fast for what it does. The IMS software environment can be divided into five main parts:

1. Database
2. Data Language I (DL/I)
3. DL/I control blocks
4. Data communications component (IMS TM)
5. Application programs

13.2 Database History

Before the development of DBMSs, data was stored in individual files. With this system, each file was stored in a separate data set in a sequential or indexed format. To retrieve data from the file, an application had to open the file and read through it to the location of the desired data. If the data was scattered through a large number of files, data access required a lot of opening

and closing of files, creating additional input/output (I/O) and processing overhead.

To reduce the number of files accessed by an application, programmers often stored the same data in many files. This practice created redundant data and the related problems of ensuring update consistency across multiple files. To ensure data consistency, special cross-file update programs had to be scheduled following the original file update.

The concept of a database system resolved many data integrity and data duplication issues encountered in a file system. A properly designed database stores the data only once in one place and makes it available to all application programs and users. At the same time, databases provide security by limiting access to data. The user's ability to read, write, update, insert, or delete data can be restricted. Data can also be backed up and recovered more easily in a single database than in a collection of flat files.

Database structures offer multiple strategies for data retrieval. Application programs can retrieve data sequentially or (with certain access methods) go directly to the desired data, reducing I/O and speeding up data retrieval. Finally, an update performed on part of the database is immediately available to other applications. Because the data exists in only one place, data integrity is more easily ensured.

The IMS database management system as it exists today represents the evolution of the hierarchical database over many years of development and improvement. IMS is in use at a large number of business and government installations throughout the world. IMS is recognized for providing excellent performance for a wide variety of applications and for performing well with databases of moderate to very large volumes of data and transactions.

13.2.1 DL/I

Because they are implemented and accessed through use of the DL/I, IMS databases are sometimes referred to as DL/I databases. DL/I is a command-level language, not a database management system. DL/I is used in batch and online programs to access data stored in databases.

Application programs use DL/I calls to request data. DL/I then uses system access methods, such as virtual storage access method (VSAM), to handle the physical transfer of data to and from the database. IMS databases are often referred to by the access method they are designed for, such as HDAM (hierarchical direct access method), HIDAM (hierarchical indexed direct

access method), PHDAM (partitioned HDAM), PHIDAM (partitioned HIDAM), HISAM (hierarchical indexed sequential access method), and SHISAM (simple HISAM). These are all IBM terms from their mainframe database products and will not be discussed here.

IMS makes provisions for nine types of access methods, and you can design a database for any one of them. On the other hand, SQL programmers are generally isolated from the access methods that their database engine uses. We will not worry about the details of the access methods that are called at this level.

13.2.2 Control Blocks

When you create an IMS database, you must define the database structure and how the data can be accessed and used by application programs. These specifications are defined within the parameters provided in two control blocks, also called DL/I control blocks:

- Database description (DBD)
- Program specification block (PSB)

In general, the DBD describes the physical structure of the database, and the PSB describes the database as it will be seen by a particular application program. The PSB tells the application which parts of the database it can access and the functions it can perform on the data. Information from the DBD and PSB is merged into a third control block, the application control block (ACB). The ACB is required for online processing but is optional for batch processing.

13.2.3 Data Communications

The IMS Transaction Manager (IMS TM) is a separate set of licensed programs that provide access to the database in an online, real-time environment. Without the TM component, you would be able to process data in the IMS database in a batch mode only.

13.2.4 Application Programs

The data in a database is of no practical use to you if it sits in the database untouched. Its value comes in its use by application programs in the performance of business or organizational functions. With IMS databases,

application programs use DL/I calls embedded in the host language to access the database. IMS supports batch and online application programs. IMS supports programs written in ADA, Assembler, C, C++, COBOL, PL/I, Pascal, REXX, and WebSphere Studio Site Developer Version 5.0.

13.2.5 Hierarchical Databases

In a hierarchical database, data is grouped in records, which are subdivided into a series of segments. Consider a department database for a school in which a record consists of the segments `Dept`, `Course`, and `Enroll`. In a hierarchical database, the structure of the database is designed to reflect logical dependencies—certain data is dependent on the existence of certain other data. Enrollment is dependent on the existence of a course, and, in this case, a course is dependent on the existence of a department to offer that course. These are called strong and weak entities in RDBMS.

The terminology changes from the SQL world to the IMS world. IMS uses records and fields, and calls each hierarchy a database. In the SQL world, a row and column can be virtual, have defaults, and have constraints—they are smart. Records and fields are physical and depend on the application programs to give them meaning—they are dumb. In SQL, a schema or database is a collection of related tables, which might map into several different IMS hierarchies in the same data model. In other words, an IMS database is more like a table in SQL.

13.2.6 Strengths and Weaknesses

In a hierarchical database, the data relationships are defined by the storage structure. The rules for queries are highly structured. It is these fixed relationships that give IMS extremely fast access to data when compared to an SQL database when the queries have not been highly optimized.

Hierarchical and relational systems have their strengths and weaknesses. The relational structure makes it relatively easy to code ad-hoc queries. But an SQL query often makes the engine read through an entire table or series of tables to retrieve the data. This makes searches slower and more processing-intensive. In addition, because the row and column structure must be maintained throughout the database, an entry must be made under each column for every row in every table, even if the entry is only a place holder (i.e., `NULL`) entry.

With the hierarchical structure, data requests or segment search arguments

(SSAs) may be more complex to construct. Once written, however, they can be very efficient, allowing direct retrieval of the data requested. The result is an extremely fast database system that can handle huge volumes of data transactions and large numbers of simultaneous users. Likewise, there is no need to enter placeholders where data is not being stored. If a segment occurrence isn't needed, it isn't created or inserted.

The trade-offs are the simplicity, portability, and flexibility of SQL versus the speed and storage savings of IMS. You tune an IMS database for one set of applications.

13.3 Simple Hierarchical Database

To illustrate how the hierarchical structure looks, we'll design two very simple databases to store information for the courses and students in a college. One database will store information on each department in the college, and the second will contain information on each college student. In a hierarchical database, an attempt is made to group data in a one-to-many relationship.

An attempt is also made to design the database so that data that is logically dependent on other data is stored in segments that are hierarchically dependent on the data. For that reason, we have designated `Dept` as the key, or root, segment for our record, because the other data would not exist without the existence of a department. We list each department only once. We provide data on each course in each department. We have a segment type `Course`, with an occurrence of that type of segment for each course in the department. Data on the course title, description, and instructor is stored as fields within the `Course` segment. Finally, we have added another segment type, `Enroll`, which will include the student IDs of the students enrolled in each course.

Notice that we are in violation ISO-11179 naming rules. The navigational model works on one instance of a data element at a time, like a magnetic tape or punch-card file, and not on whole sets like RDBMS.

In Figure 13.1, we also created a second database called `Student`. This database contains information on all the students enrolled in the college. This database duplicates some of the data stored in the `Enroll` segment of the `Department` database. Later, we will construct a larger database that eliminates the duplicated data. The design we choose for our database depends on a number of factors; in this case, we will focus on which data we will need to access most frequently.

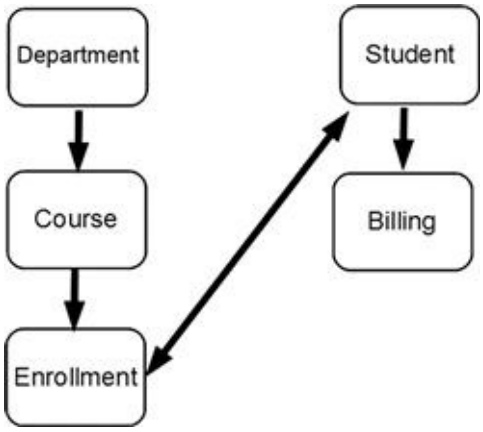


FIGURE 13.1 Sample hierarchical databases for `Department` and `Student`.

The two databases in the figure are shown as they might be structured in relational form in three tables. Notice that these tables are not always normalized when you do a direct translation to SQL. For example:

```
CREATE SCHEMA College;
CREATE TABLE Courses
(course_nbr CHAR(9) NOT NULL PRIMARY KEY,
  course_title VARCHAR(20) NOT NULL,
  course_description VARCHAR(200) NOT NULL,
  dept_id CHAR(7) NOT NULL
  REFERENCES Departments (dept_id)
  ON UPDATE CASCADE);
CREATE TABLE Students
(student_id CHAR(9) NOT NULL PRIMARY KEY,
  student_name CHAR(35) NOT NULL,
  student_address CHAR(35) NOT NULL,
  major CHAR(10));
CREATE TABLE Departments
(dept_id CHAR(7) NOT NULL PRIMARY KEY,
  dept_name CHAR(15) NOT NULL,
  chairman_name CHAR(35) NOT NULL,
  budget_code CHAR(3) NOT NULL);
```

13.3.1 `Department` Database

The segments in the `Department` database are as follows:

- `Dept`: Information on each department. This segment includes fields for the department ID (key field), department name, chairman's name, number of faculty, and number of students registered in departmental courses.
- `Course`: This segment includes fields for the course number (a unique identifier), course title, course description, and instructor's name.
- `Enroll`: The students enrolled in the course. This segment includes fields for student ID (key field), student name, and grade.

13.3.2 `Student` Database

The segments in the `Student` database are as follows:

- `Student`: Student information. This segment includes fields for student ID (key field), student name, address, major, and courses completed.
- `Billing`: Billing information for courses taken. This segment includes fields for semester, tuition due, tuition paid, and scholarship funds applied.

The double-headed line between the root (`Student`) segment of the `Student` database and the `Enroll` segment of the `Department` database represents a logical relationship based on data residing in one segment and needed in the other. This is not like the referencing and referenced table structures in SQL; it has to be enforced by the application programs.

13.3.3 Design Considerations

Before implementing a hierarchical structure for your database, you should analyze the end user's processing requirements, because they will determine how you structure the database. In particular, you must consider how the data elements are related and how they will be accessed.

For example, given the classic `Parts and Suppliers` database, the hierarchical structure could subordinate parts under suppliers for the accounts receivable department, or subordinate suppliers under parts for the order department. In RDBMS, there would be a relationship table that references both parts and suppliers by their primary keys and that contains information that pertains to the relationship, and not to either parts or suppliers.

13.3.4 Example Database Expanded

At this point you have learned enough about database design to expand our original example database. We decide that we can make better use of our college data by combining the `Department` and `Student` databases. Our new `College` database is shown in Figure 13.2.

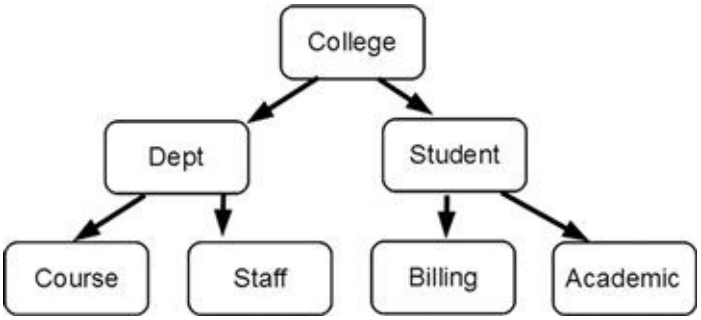


FIGURE 13.2 `College` database (combining `Department` and `Student` databases).

The following segments are in the expanded `College` database:

- `College`: The root segment. One record will exist for each college in the university. The key field is the college ID, such as `ARTS`, `ENGR`, `BUSADM`, and `FINEARTS`.
- `Dept`: Information on each department within the college. This segment includes fields for the department ID (key field), department name, chairman's name, number of faculty, and number of students registered in departmental courses.
- `Course`: This segment includes fields for the course number (key field), course title, course description, and instructor's name.
- `Enroll`: A list of students enrolled in the course. There are fields for student ID (key field), student name, current grade, and number of absences.
- `Staff`: A list of staff members, including professors, instructors, teaching assistants, and clerical personnel. The key field is employee number. There are fields for name, address, phone number, office number, and work schedule.
- `Student`: Student information. This segment includes fields for student ID (key field), student name, address, major, and courses being taken currently.
- `Billing`: Billing and payment information. It includes fields for billing date (key field), semester, amount billed, amount paid, scholarship funds

applied, and scholarship funds available.

- `Academic`: The key field is a combination of the year and semester. Fields include grade point average (GPA) per semester, cumulative GPA, and enough fields to list courses completed and grades per semester.

13.3.5 Data Relationships

The process of data normalization helps you break data into naturally associated groupings that can be stored collectively in segments in a hierarchical database. In designing your database, break the individual data elements into groups based on the processing functions they will serve. At the same time, group data based on inherent relationships between data elements.

For example, the `College` database (see Figure 13.2) contains a segment called `Student`. Certain data is naturally associated with a student, such as student ID number, student name, address, and courses taken. Other data that wanted in the `College` database, such as a list of courses taught or administrative information on faculty members, would not work well in the `Student` segment.

Two important data relationship concepts are one-to-many and many-to-many. In the `College` database, there are many departments for each college, but only one college for each department. Likewise, many courses are taught by each department, but a specific course (in this case) can be offered by only one department.

The relationship between courses and students is many-to-many, as there are many students in any course and each student will take several courses. Let’s ignore the many-to-many relationship for now—this is the hardest relationship to model in a hierarchical database.

A one-to-many relationship is structured as a dependent relationship in a hierarchical database: the many are dependent on the one. Without a department, there would be no courses taught; without a college, there would be no departments.

Parent and child relationships are based solely on the relative positions of the segments in the hierarchy, and a segment can be a parent of other segments while serving as the child of a segment above it. In Figure 13.2, `Enroll` is a child of `Course`, and `Course`, although the parent of `Enroll`, is also the child of `Dept`. `Billing` and `Academic` are both children of `Student`, which is a child of `College`. Technically, all of the segments except `College`

are dependents.

When you have analyzed the data elements, grouped them into segments, selected a key field for each segment, and designed a database structure, you have completed most of your database design. You may find, however, that the design you have chosen does not work well for every application program. Some programs may need to access a segment by a field other than the one you have chosen as the key. Or another application may need to associate segments that are located in two different databases or hierarchies. IMS has provided two very useful tools that you can use to resolve these data requirements: secondary indexes and logical relationships.

Secondary indexes let you create an index based on a field other than the root segment key field. That field can be used as if it were the key to access segments based on a data element other than the root key.

Logical relationships let you relate segments in separate hierarchies and, in effect, create a hierarchic structure that *does not actually exist in storage.* The logical structure can be processed as if it physically exists, allowing you to create logical hierarchies without creating physical ones.

13.3.6 Hierarchical Sequence

Because segments are accessed according to their sequence in the hierarchy, it is important to understand how the hierarchy is arranged. In IMS, segments are stored in a top-down, left-to-right sequence (Figure 13.3). The sequence flows from the top to the bottom of the leftmost path or leg. When the bottom of that path is reached, the sequence continues at the top of the next leg to the right.

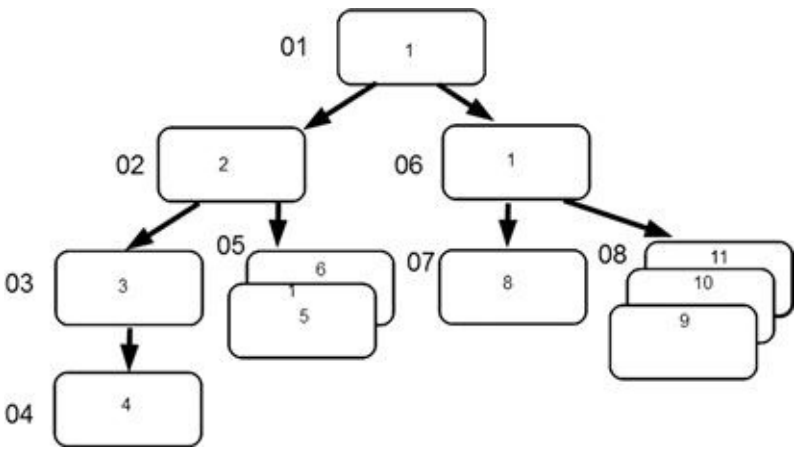


FIGURE 13.3 Sequence and data paths in a hierarchy.

Understanding the sequence of segments within a record is important to

understanding movement and position within the hierarchy. Movement can be forward or backward and always follows the hierarchical sequence. Forward means from top to bottom, and backward means bottom to top. Position within the database means the current location at a specific segment. You are once more doing depth-first tree traversals, but with a slightly different terminology.

13.3.7 Hierarchical Data Paths

In Figure 13.4, the numbers inside the segments show the hierarchy as a search path would follow it. The numbers to the left of each segment show the segment types as they would be numbered by type, not occurrence. That is, there may be any number of occurrences of segment type 04, but there will be only one type of segment 04. The segment type is referred to as the *segment code*.

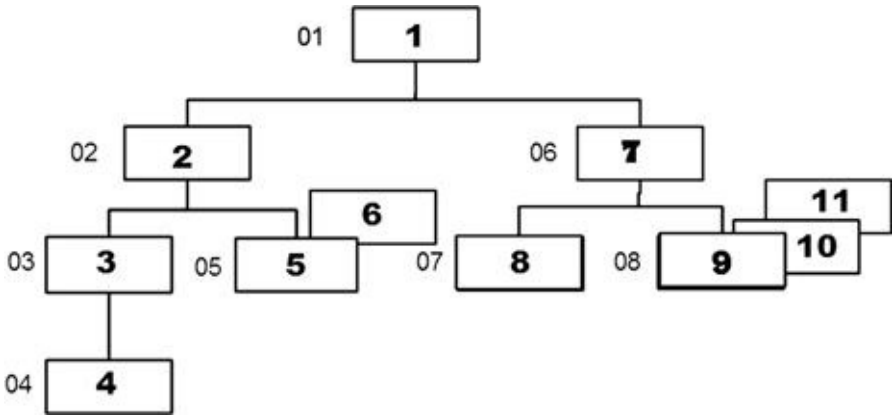


FIGURE 13.4 Hierarchical Data Paths.

To retrieve a segment, count every occurrence of every segment type in the path and proceed through the hierarchy according to the rules of navigation:

- Top to bottom
- Front to back (counting twin segments)
- Left to right

For example, if an application program issues a `GET-UNIQUE` (GU) call for segment 6 in Figure 13.4, the current position in the hierarchy is immediately following segment 06. If the program then issues a `GET-NEXT` (GN) call, IMS would return segment 07. There is also the `GET-NEXT WITHIN PARENT` (GNP) call, which explains itself.

As shown in Figure 13.4, the `College` database can be separated into four search paths:

- The first path includes segment types 01, 02, 03, and 04.
- The second path includes segment types 01, 02, and 05.
- The third path includes segment types 01, 06, and 07.
- The fourth path includes segment types 01, 06, and 08. The search path always starts at 01, the root segment.

13.3.8 Database Records

Whereas a database consists of one or more database records, a database record consists of one or more segments. In the `College` database, a record consists of the root segment `College` and its dependent segments. It is possible to define a database record as only a root segment. A database can contain only the record structure defined for it, and a database record can contain only the types of segments defined for it.

The term *record* can also be used to refer to a data set record (or block), which is not the same thing as a database record. IMS uses standard data system management methods to store its databases in data sets. The smallest entity of a data set is also referred to as a *record* (or block).

Two distinctions are important. A database record may be stored in several data set blocks. A block may contain several whole records or pieces of several records. In this chapter, I try to distinguish between a database record and data set record where the meaning may be ambiguous.

13.3.9 Segment Format

A segment is the smallest structure of the database in the sense that IMS cannot retrieve data in an amount less than a segment. Segments can be broken down into smaller increments called *fields*, which can be addressed individually by application programs.

A database record can contain a maximum of 255 types of segments. The number of segment occurrences of any type is limited only by the amount of space you allocate for the database. Segment types can be of fixed length or variable length. You must define the size of each segment type.

It is important to distinguish the difference between segment types and segment occurrences. `Course` is a type of segment defined in the DBD for the `College` database. There can be any number of occurrences for the `Course` segment type. Each occurrence of the `Course` segment type will be exactly as

defined in the DBD. The only difference in occurrences of segment types is the data contained in them (and the length, if the segment is defined as variable length).

Segments have several different possible structures, but from a logical viewpoint, there is a prefix that has structural and control information for the IMS system, and 3 is the prefix for the actual data fields.

In the data portion, you can define the following types of fields: a sequence field and the data fields. The *sequence field* is often referred to as the *key field*. It can be used to keep occurrences of a segment type in sequence under a common parent, based on the data or value entered in this field. A key field can be defined in the root segment of a HISAM, HDAM, or HIDAM database to give an application program direct access to a specific root segment. A key field can be used in HISAM and HIDAM databases to allow database records to be retrieved sequentially. Key fields are used for logical relationships and secondary indexes.

The key field not only can contain data but also can be used in special ways that help you organize your database. With the key field, you can keep occurrences of a segment type in some kind of key sequence, which you design. For instance, in our example database you might want to store the student records in ascending sequence, based on student ID number. To do this, you define the student ID field as a unique key field. IMS will store the records in ascending numerical order. You could also store them in alphabetical order by defining the name field as a unique key field. Three factors of key fields are important to remember:

1. The data or value in the key field is called the *key* of the segment.

2. The key field can be defined as unique or nonunique.

3. You do not have to define a key field in every segment type

You define *data fields* to contain the actual data being stored in the database. (Remember that the sequence field is a data field.) Data fields, including sequence fields, can be defined to IMS for use by applications programs.

13.3.10 Segment Definitions

In IMS, segments are defined by the order in which they occur and by their relationship with other segments:

- *Root segment:* The first, or highest, segment in the record. There can be only one root segment for each record. There can be many records in a database.
- *Dependent segment:* All segments in a database record except the root segment.
- *Parent segment:* A segment that has one or more dependent segments beneath it in the hierarchy.
- *Child segment:* A segment that is a dependent of another segment above it in the hierarchy.
- *Twin segment:* A segment occurrence that exists with one or more segments of the same type under a single parent.

There are functions to edit, encrypt, or compress segments, which we will not consider here. The point is that you have a lot of control of the data at the physical level in IMS.

13.4 Summary

> *"Those who cannot remember the past are condemned to repeat it." — George Santayana*

There were databases before SQL, and they were all based on a navigation model. What SQL programmers do not like to admit is that not all commercial information resides in SQL databases. The majority is still in simple files or older, navigational, nonrelational databases.

Even after the new tools have taken on their own characteristics to become a separate species, the mental models of the old systems still linger. The old patterns are repeated in the new technology.

Even the early SQL products fell into this trap. For example, how many SQL programmers today use `IDENTITY`, or other auto-increment vendor extensions as keys on SQL tables today, unaware that they are imitating the navigational sequence field (a.k.a. the key field) from IMS?

This is not to say that a hierarchy is not a good way to organize data; it is! But you need to see the abstraction apart from any particular implementation. SQL is a declarative language, while DL/I is a collection of procedure calls inside a host language. The temptation is to continue to write SQL code in the same style as you wrote procedural code in COBOL, PL/I, or whatever host language you had.

The bad news is that you can use cursors to imitate sequential file routines. Roughly, the `READ()` command becomes an embedded `FETCH` statement, `OPEN` and `CLOSE` file commands map to `OPEN CURSOR` and `CLOSE CURSOR` statements, and every file becomes a simple table without any constraints and a "record number" of some sort. The conversion of legacy code is almost effortless with such a mapping. And it is also the worst way to program with a SQL database.

Hopefully, this book will show you a few tricks that will let you write SQL as SQL and not fake a previous language in it.

Concluding Thoughts

IMS is almost 50 years old and has moved from the early IBM System/360 technology and COBOL to LINUX and Java. Failure to understand how all that data is modeled and accessed means you cannot get to that data. My personal experience was being teased by younger programmers for not knowing the current cool application programming language. When I asked them what they were doing, they were moving IBM 3270 terminal applications onto cell phones; the back end was an IMS database that had been at their insurance company since 1970. As Alphonse Karr said in 1839: "*Plus ça change, plus c'est la même chose*"—"the more it changes, the more it's the same thing."

References

1. The www.dbazine.com website has a detailed three-part tutorial on IMS, from which this material was brutally extracted and summarized.
2. The best source for IMS materials is at http://www.redbooks.ibm.com/ where you can download manuals directly from IBM.
3. Dutta A. *IMS concepts and database administration*. 2010, 2012; Houston, TX. *https://communities.bmc.com/docs/DOC-9908*; 2010, 2012.
4. Meltz D, et al. *An introduction to IMS: Your complete guide to IBM's information management system*. Upper Saddle River, NJ: IBM Press (Pearson Education); 2005.

Glossary

ACID: Atomicity, consistency, isolation, and durability, the four desirable properties of a classic transaction processing system. Each transaction is an atomic (indivisible) unit of work that fails or succeeds as a unit. The database is always in a consistent state at the start and end of a transaction; no constraints are violated. Each transaction is isolated from all the other transactions against the database. Finally, the work done by a transaction is persisted in the database (durable) when a transaction succeeds. *See* BASE.

Array: A data structure that uses one or more numeric position indexes (subscripts) to locate a value. The elements of an array are all of the same data type.

BASE: A deliberately cute acronym that is short for basically available, soft state, eventual consistent. This is the "no SQL" counterpart to the ACID property of RDBMS. The idea is that we can live with a system that is known to be incomplete and inaccurate, under the assumption that it will "catch up" with the truth. We are willing to wait to get data (basically available), can live with different users seeing slightly inconsistent versions of the same data (soft state), and believe the data to eventually arrive at a consistent state. The trade-off is performance for accuracy.

Batch processing: A technique of submitting a large set of transactions as a single unit of work. Typically in databases, this is how large amounts of data are inserted, updated, or deleted from the database.

Bertillon card: An early biometric identification system from France based on an elaborate system of body measurements. It was used for identifying criminals. It has been replaced by fingerprints, DNA, and other biometric identifiers.

Big Data: A current buzzword usually associated with NoSQL and mixed data sources. Forrester Research created a definition with the catchy buzz phrase "the 4 V's" in their literature. The first V is volume, which has been growing at an exponential rate. The second V is velocity. Data arrives faster than it has before, thanks to improved communication systems such as fiber-optic networks. The third V is variety. The sources

of data have increased because the variety of devices and their applications. For example, social networks, blogs, and web cameras did not exist in the past. The fourth V is variability. There is an increasing variety of data formats, not just relational or traditional data stores. Others have tried to add more V's to the Big Data definition: verification, value, veracity, and vicinity are some of the candidate buzzwords.

Biometrics: Data based on human physical measurements such as fingerprints, retina prints, and so forth. *See* Bertillon card.

Brewer's theorem: Named for Eric Brewer who proposed it at the 2000 Symposium on Principles of Distributed Computing (PODC). *See* CAP.

Byte: A unit of computer storage made of 8 bits (binary digits). The term was coined by Werner Buchholz in July 1956, during the early design phase for the IBM Stretch computer. There are several ways to express the size of computer storage using the base unit of 1,024 bits = 1 kilobyte or 1 KB, and copying the SI prefixes for decimal powers to apply as powers of the base unit.

1,024	Kbytes	kilobytes
$1,024^2$	Mbytes	megabytes
$1,024^3$	Gbytes	gigabytes
$1,024^4$	Tbytes	terabytes
$1,024^5$	Pbytes	petabytes
$1,024^6$	Ebytes	exabytes
$1,024^7$	Zbytes	zettabytes
$1,024^8$	Ybytes	yottabytes

As of 2013, there are several commercial databases that are measured in petabytes, such as Wal-Mart's data warehouse. Oracle has used the prefix "exa-" in their advertising for their data warehouse product, but has no installations that actually use that unit. The prefix "yotta-" apparently sounds too silly to be used for advertising.

CAP: This was a conjecture made by Eric Brewer that stands for consistency, availability, and partition tolerance as a transaction model for distributed databases. The CAP theorem states that it is impossible for a distributed computer system to simultaneously provide all three of the following system characteristics:

- Consistency (all nodes in the distributed database see the same data at the same time).

- ◆ Availability (every database request receives a response about its success or failure).
- ◆ Partition tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system).

In 2002, Seth Gilbert and Nancy Lynch of MIT published a formal proof of Brewer's conjecture. *See* Brewer's theorem.

Cloud computing: The practice of using a network of remote servers hosted on the Internet to store, manage, and process data, rather than a local server. Cloud computing is mostly jargon that is used for any distributed computing over a network.

CODASYL: The committee that defined COBOL and came up with a standard for the navigational database model. Their work with databases was taken over by the ANSI X3H2 Database Standards Committee. This committee took the CODASYL model, formalized it a bit, and produced the NDL (Network Database Language) specification. The NDL standard simply expired from lack of interest once the SQL standards were published.

Codd: Edgar Frank "Ted" Codd (d. 2003) was the inventor the relational model for database management, the theoretical basis for relational databases. His mathematical approach influenced all database models that followed it. In particular, he published his 12 rules of RDBMS to define what constituted a relational database. Dr. Codd later coined the term online analytical processing (OLAP) and wrote the 12 laws of OLAP.

Columnar databases: A relational database, usually SQL, that stores the data in columns rather than in rows. One advantage is that a column is of one and only one data type so it can be compressed and these columnar stores can be used to create many different tables.

`COMMIT` **statement:** A statement that persists the data in a user session into the database. This was first used in databases that have an ACID model, but applies to any database. In an ACID model, the persisted data has to meet all the constraints and be immediately consistent; in a BASE model, the data is simply persisted and we wait to find out if it is consistent. *See* `ROLLBACK` statement and `SAVEPOINT` statement.

Complex event processing (CEP): Means that not all the data has arrived in the database yet! You cannot yet complete a query because you are anticipating data, but you know you have part of it. The data can be from the same source, or from multiple sources. The event model is not quite

the same as a state transition model for a system. State transitions are integrity checks that assure data changes only according to rules for sequences of procedures, of fixed or variable lifespans.

Compression: Storing data in more compact format to save disk space and speed up data transfers from secondary to primary storage. Compression can lose some information from data from the original for some data types, such as music or graphics, or it can be lossless. This means the original data can be completely reconstructed from the compressed data.

Connected graph: From graph theory, a set of nodes in which any two nodes can be reached by a walk. This is a nice property for queries since any two nodes can be related (though the relationship might be insanely distant).

Consistency: The database property of persisting only data that meets all of the constraints, implicit or explicit, in the schema.

CQL: Contextual Query Language. A string pattern searching language for textbases defined by ANSI Z39. It is based on regular expressions. It is similar to commercial text search languages such as LexisNexus and Westlaw. *See* Regular expressions and Textbase

Cycle: A path in a graph that returns to the node from which it started. Also known as *circuit* in graph theory. A Hamiltonian circuit is one that contains all nodes in a graph. In RDBMS, it refers to circular references among referential actions; it is not desirable because data retrieval can hang in an endless loop.

Cypher: A declarative graph query language that is still growing and maturing, which will make SQL programmers comfortable.

DCL: Data Control Language. One of the SQL sublanguages that controls user access to the schema. It is not a security or encryption tool. The term is now applied to non-SQL databases as well.

DDL: Data Definition Language. One of the SQL sublanguages that describes and modifies the tables, views, indexing, procedures, and other schema objects. The term is now applied to non-SQL databases as well.

DML: Data Manipulation Language. One of the SQL sublanguages that performs queries, invokes procedures, and updates the data in the schema. It does not change the structure of the schema or control access. The term is now applied to non-SQL databases as well.

DNA (Deoxyribonucleic Acid): The chemical basis of genes and

chromosomes that makes each human being unique. DNA profiling is used for identification by encoding repetitive (repeat) gene sequences that are highly variable. These easily classified units of genetic material are called variable number tandem repeats (VNTRs) and, particularly, short tandem repeats (STRs). DNA profiling is not full genome sequencing.

Document management systems: *See* Textbase.

Edge: From graph theory, also known as an arc. They are the parts of a graph structure that connect two nodes. The edges can be undirected or directed to show if the relationship is asymmetric or symmetric, respectively. They can also have values, such as distances on a graph that models a physical map.

Face recognition: A family of biometric recognition algorithms can be divided into two main approaches: geometric or photometric. The geometric family assigns points on a human face and constructs a geometric model based on the distances and ratios among those points. This mesh can be encoded easily. The photometric family attempts to align a photographic image of a human face with base image from a database.

Four V's: *See* Big Data.

Galton fingerprint system: Also Henry–Galton. One of several fingerprint classifications systems, based the physical appearance of fingerprints. It is used in the United States and many optical matching systems use it.

Generational concurrency model: A concurrency control model that is based on having a "snapshot" of the state of a database at a point in time (a generation) when the database was consistent. The database can be queried with or restored to that snapshot. This model was derived from microfilm systems that printed multiple copies of the same record for multiple users. Updates occur only when the various updates to the outstanding copies can be combined into a single consistent update. *See* Optimistic concurrency.

Geographic information system (GIS): A database specialized for geographical or spatial data. There are geospatial standards from ISO Technical Committee 211 (ISO/TC 211) and the Open Geospatial Consortium (OGC). The OGC is an international industry group. Today, the big players are Environmental Systems Research Institute (ESRI), Computer-Aided Resource Information System (CARIS), Mapping Display and Analysis System (MIDAS, now MapInfo), and Earth

Resource Data Analysis System (ERDAS). There are also two public domain systems (MOSS and GRASS GIS) that began in the late 1970s and early 1980s.

Graph: A mathematical structure that consists of nodes (a.k.a. vertices) connected by edges (a.k.a. arcs). Each edge connects zero nodes, one node to itself, or two different nodes. A node is usually drawn as a circle or dot and the edges are drawn as lines in diagrams. *See* Connected graph, Edge, Node, Path, Tree, and Walk.

Graph databases: A database stores relationship data in a graph structure. They are not good for computations and aggregations. *See* Edge, Graph, Gremlin, Neo4j, and Node.

Gremlin: An open-source language that is based on traversals of a property graph with a syntax taken from OO and the C programming language family.

Hashing: A data access method that takes a search key and applies a mathematical function to it to get a position in a hash table. This hash table is a lookup table that holds the actual physical location of the data in physical storage. It is possible that two different keys can return the same hash value (a collision or hash clash) and these have to be resolved in some way. A perfect hashing function has no collisions.

HDFS: The Hadoop Distributed File System is built from commodity hardware arranged to be fault-tolerant. The logo for this software is a cute cartoon baby elephant. HDFS and MongoDB are the most popular versions of NoSQL databases.

Hierarchical database systems: A prerelational family of databases based on accessing data with a hierarchical data structure. It is a special case of the network or navigational family of databases. *See* IMS and Network database systems.

Hierarchical triangular mesh (HTM): A geographic location system based on a recursive geodesic covering of Earth with triangles called trixels. The smallest valid HTM ID is 8 levels but it is easy to go to 31 levels, represented in 64 bits. Level 25 is good enough for most applications, since it is about 0.6 meters on the surface of the Earth, or 0.02 arc-seconds. Level 26 is about 30 centimeters (less than one foot) on the Earth's surface.

Hive: This is an open-source Hadoop language from Facebook. It is closer to SQL than Pig and can be used for ad-hoc queries without being

compiled like Pig.

Information management system (IMS): An IBM product that is the most popular hierarchical database in use today. Information is structured in records that are subdivided into a hierarchical tree of related segments. A record is a root segment and all of its dependent segments. Segments are further subdivided into fields. The data in any record relates to one entity. It is stable, well defined, scalable, and very fast for what it does. IMS databases are often referred to by the access method they are designed for, such as HDAM (hierarchical direct access method), HIDAM (hierarchical indexed direct access method), PHDAM (partitioned HDAM), PHIDAM (partitioned HIDAM), HISAM (hierarchical indexed sequential access method), and SHISAM (simple HISAM).

Isolation levels: The scheme by which a database decides how a session see modifications made to the database by other sessions. There is an ANSI/ISO standard SQL model that is based on committed and uncommitted work and the ACID properties. *See* BASE, as an alternative.

Java: A programming language currently owned by the Oracle Corporation. The JDBC (Java Database Connectivity) is a Java-based data access technology that defines how a client may access a database. JDBC is oriented toward relational databases, but has become the most popular application program interface for most of the NoSQL products.

Key–Value stores: A model of data storage that has a physical locator (key) and a value. These pairs are searched on the key with hashing or indexes and the value is returned. *See* MapReduce.

Keyword and keyword in context (KWIC): A family of text indexing techniques that look for a searched or keyword and return its location in a textbase. The classic KWIC displays the keyword in bold type, with words to left and right of it in regular type. The family includes KWOC (keyword out of context), KWAC (keyword augmented in context), and key-term alphabetical (KEYTALPHA). The differences in these techniques are in the display to the user.

LAMP stack: An architecture for websites based on open-source software. The initials stand for Linux (operating system), Apache (HTTP server), MySQL (database, but since it was acquired by Oracle, people are moving to the open-source version, MariaDB), and PHP, Perl, or Python for the application language.

MapReduce: A data access technique for large data sets that depends on

parallelism in the storage used. First, a `Map()` procedure does filtering and sorting to get the data into queues, then a `Reduce()` procedure performs summary and aggregations with data drawn from the queues. This model is based on the map and reduce functions used in functional programming languages, such as LISP.

Master street address guide (MSAG): The MSAG describes address elements including the exact spellings of street names and street number ranges. It is part of the U.S. Postal Service's (USPS) Coding Accuracy Support System (CASS). There are similar guides in other countries.

MDX: A programming language for OLAP queries from Microsoft that has become a de-facto standard by virtue of Microsoft's market domination.

MOLAP: Multidimensional online analytical processing. This is the "data in a grid" version of OLAP that is preferred by spreadsheet users. *See* OLAP and ROLAP.

MongoDB: An open-source document-oriented database system developed and supported by 10gen. MongoDB stores structured data as documents in the BSON format, a version of JSON (JavaScript Object Notation). MongoDB is the most popular NoSQL database management system and has many third-party tools.

Multivalued databases: These databases are also called NFNF, 2NF, NF2, and ¬NF in the literature. They do not follow first normal form (1NF) by allowing an unordered list of values in a column of a table. They require a set of operators to nest and unnest these structures in addition to the usual relational operators with appropriate extensions.

Navigational databases: *See* Network databases and IMS.

Neo4j: The most popular graph programming language.

Network databases: A family of prerelational databases associated with Charles Bachman, who did pioneering in the 1950s at Dow Chemical and in the 1960s at General Electric, where he developed the Integrated Data Store (IDS), one of the first database management systems. The network and hierarchical models are called *navigational* databases because the mental model of data access is that of a reader moving along paths to pick up the data. When Bachman received the ACM Turing Award in 1973 for his outstanding contributions to database technology, this is how he described it. ANSI X3H2 produced a short-lived standard for NDL (Network Database Language) that was never implemented.

NFNF databases: Nonfirst normal form. These databases are also called NFNF, 2NF, NF2, and ¬NF in the literature. *See* Multivalued databases.

NIST: National Institute for Science and Technology. An agency of the U.S. federal government that sets IT and other standards in the United States. It began life as the Bureau of Weights and Measures in the 1700 s.

Node: Nodes are also called vertices. The part of a graph structure that is connected by edges. The nodes usually model an entity or element of a relationship. For example, if we use a graph for a schematic subway map, the nodes would be the subway stations connected by rails.

NoSQL: A buzzword that has be defined as “no sequel” or (better) “not only SQL” in the literature. It is usually applied to MapReduce databases, but has a more general (and vague) meaning.

OLAP: Online analytical processing. These products use a snapshot of a database taken at one point in time and provide analysis against the data. This lead to hybrid OLAP, which retains some of result tables in specialized storage or indexing so that they can be reused. The base tables, dimension tables, and some summary tables are in the RDBMS. This is probably the most common approach in products today. *See* ROLAP and MOLAP.

OLTP: Online transaction processing. Its purpose is to provide support for daily business applications. This is the niche that SQL has in the commercial market.

Optimistic concurrency: A concurrency control model for databases based on the assumption that multiple users will seldom want to modify the same data at the same time. When conflicts are detected, the actions are rolled back to a previous consistent database state. This is also called generational concurrency because the prior database states are retained. *See* Pessimistic concurrency.

Optimizer: The part of a database engine that attempts to pick the best statement execution plan from many possible equivalent plans. This is one way that declarative programming (“tell the machine what you want”) differs from procedural programming (“tell the machine exactly how to do it”). The same statement can have different execution plans because the optimizer will use the current state of the database (indexing, statistics, cached data from other sessions, etc.) at the time of invocation.

Path: From graph theory, a path is a walk that goes through each node only once. If you have n nodes, you will have $(n - 1)$ edges in the path.

Pessimistic concurrency: A concurrency control model for databases based on the assumption that multiple users will always want to modify the same data at the same time. Records have to have locks to prevent this. *See* Optimistic concurrency.

Petri nets: A mathematical modeling tool that uses a graph that can hold and move tokens in its nodes. Petri nets are used to model concurrency problems in networks.

Pick: This product began life as the Generalized Information Retrieval Language System (GIRLS) on an IBM System/360 in 1965. It is still in use and has become a generic name for this type of database. It is a classic NFNF data model and uses a language that evolved from a version of BASIC.

Pig Latin: Or simply Pig, is a query language developed by Yahoo and is now part of the Hadoop project.

Query: A statement that returns a result set from a database without changing the data. A query does not need to be logged for backup and restoration. However, for full audits they are needed to show who saw what data and when they saw it.

RAID: Redundant array of independent disks (originally redundant array of inexpensive disks). A family of disk storage hardware arrangements, based on the concept that if the system has redundancy, a failure can be dynamically repaired by hardware replacement without loss of database access.

Regular expressions: A string pattern matching system first developed by the mathematician Stephen Cole Kleene. It is the basis for the `grep()` family in UNIX and other programming languages. There are many vendor and language versions of this tool. ANSI/ISO standard SQL has a simple `LIKE` predicate and more complex `SIMILAR TO` predicate.

ROLAP: Relational online analytical processing. It was developed after multidimensional OLAP. The main difference is that ROLAP does not do precomputation or store summary data in the database.

`ROLLBACK` statement: A statement that restores a database to its prior state and ends a user session. This was first used in databases that have an ACID model, but applies to any database that does not persist changes immediately. *See* `COMMIT` statement and `SAVEPOINT` statement.

`SAVEPOINT` statement: A statement that sets a point in a user session so that

a transaction can be rolled back to the state of the database at which the savepoint was set. Think of it as an "almost commit," which is given a name for the rollback. This was first used in databases that have an ACID model, but applies to any database that does not persist changes immediately. *See* `COMMIT` statement and `ROLLBACK` statement.

Schema/no schema: A formal description of the data structure used in a database. In SQL, this is done with the DDL (Data Definition Language), which describes the tables, views, indexing, procedures, and so forth. Other database models may have their own schema language, or have a no schema model. The no schema model usually has metadata embedded in the same storage as the data. The most common example is the use of tags in markup languages and key–value pairs.

SMAQ stack: Pronounced "smack stack," this is an architecture for Big Data storage. The letters stand for storage, MapReduce, and query, and assume commodity hardware with open-source software. SMAQ systems are typically open source, distributed, and run on commodity hardware. This is a parallel to the commodity LAMP stack for websites. LAMP stands for Linux, Apache, MySQL, and PHP in that niche.

Sqoop: A database-agnostic tool that uses the Java JDBC database API. Tables can be imported either wholesale or using queries to restrict the data import. Sqoop also offers the ability to reinject the results of MapReduce from HDFS back into a relational database.

Streaming database: A database designed to process data that comes in a stream outside the control of the database. The classic examples are stock and commodity trades and instrument sampling. You can find products from IBM (SPADE), Oracle (Oracle CEP), Microsoft (StreamInsight), and smaller vendors, such as StreamBase (stream-oriented extension of SQL) and Kx (Q language, based on APL), as well as open-source projects (Esper, stream-oriented extension of SQL). Broadly speaking, the languages are SQL-like and readable or they are C-like and cryptic. As examples of the two extremes, look at StreamBase and Kx.

Textbase: A modern term for what we called document management systems before. They are concerned with searching large volumes of text. NISO, the National Information Standards Organization and now the ANSI Z39 group, set standards in this area. Its membership is drawn from organizations in the fields of publishing, libraries, IT, and media organizations. *See* CQL.

Tree:

- ◆ Graph databases: This is a connected graph that has no cycles.
- ◆ Indexes: An access method that uses one of many possible tree structures to build pointer chains that eventually lead to a physical record.

Unicode: A computing industry standard for the consistent encoding, representation, and handling of text expressed in most of the world's writing systems. The standard is maintained by the Unicode Consortium and uses the UTF-8 and UTF-16 representations. UTF-8 uses 1 byte for any ASCII characters, which have the same code values in both UTF-8 and ASCII encoding, and up to 4 bytes for other characters. UTF-16 uses two 16-bit units (4 × 8 bits) to handle each of the additional characters.

Walk: Graph theory. This is a sequence of edges that connect a set of nodes, without repeating an edge.

WordNet: A lexical database for the English language that puts English words into sets of synonyms called synsets. It is used by textbases for semantic searches.

X3H2: More properly ANSI X3H2, now known as INCITS H2. This is the committee that sets the SQL language and other database-related IT standards. ANSI stands for American National Standards Institute, and INCITS stands for International Committee for Information Technology Standards (it is pronounced "insights"). It was formerly known as the X3 and NCITS. The SQL standards over the years have been: SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2008, and, as of this writing, SQL:2011. ANSI/ISO standards are reviewed every five years and can be deprecated at that time.

XML: eXtensible Markup Language. A markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It is an international standard for exchanging data with strong support via Unicode for the languages of the world. Although the design of XML focuses on documents, it is widely used for the representation of arbitrary data structures. The major concept is tags that come in pairs written in < tag > and </tag > to bracket the unit of data. Other features use the angle brackets to separate them from the data.

ZIP code: The zone improvement plan (ZIP) code is a geographical encoding system used by the (USPS for addresses. The term Zip code has become a generic term for any address location code in other countries.

Index

Note: Page numbers followed by *f* indicate figures, *b* indicate boxes and *t* indicate tables.

C

E

G

H

O

R

S