JOE CELKO'S COMPLETE GUIDE TO NoSQL

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

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.

To the fullest extent of the law, neither the Publisher nor the authors, contributors, or editors, assume any liability for any injury and/or damage to persons or property as a matter of products liability, negligence or otherwise, or from any use or operation of any methods, products, instructions, or ideas contained in the material herein.

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*

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.

NoSQL and 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.

Joe Celko's Complete Guide to NoSQL. DOI: 10.1016/B978-0-12-407192-6.00001-7

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.