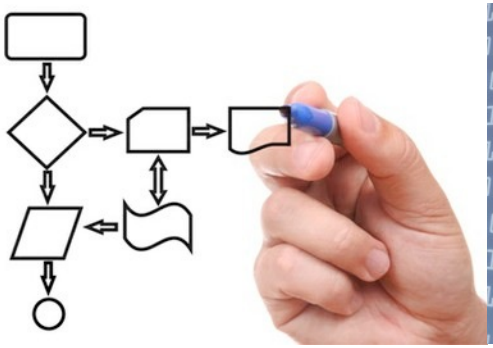
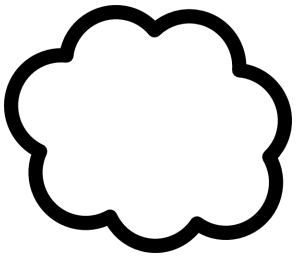
San Jose State University

CMPE 281

Cloud Technologies

Introduction to NoSQL & Map Reduce





The Papers

MapReduce: Simplified Data Processing on Large Clusters

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

jeff@google.com, sanjay@google.com

Google, Inc.

Abstract

MapReduce is a programming model and an associated implementation for processing and generating large data sets. Users specify a *map* function that processes a key/value pair to generate a set of intermediate key/value pairs, and a *reduce* function that merges all intermediate values associated with the same intermediate key. Many real world tasks are expressible in this model, as shown in the paper.

Programs written in this functional style are automatically parallelized and executed on a large cluster of commodity machines. The run-time system takes care of the details of partitioning the input data, scheduling the program's execution across a set of machines, handling machine failures, and managing the required inter-machine communication. This allows programmers without any experience with parallel and distributed systems to easily utilize the resources of a large distributed system.

Our implementation of MapReduce runs on a large cluster of commodity machines and is highly scalable: a typical MapReduce computation processes many terabytes of data on thousands of machines. Programmers find the system easy to use: hundreds of MapReduce programs have been implemented and upwards of one thousand MapReduce jobs are executed on Google's clusters every day.

1 Introduction

Over the past five years, the authors and many others at Google have implemented hundreds of special-purpose computations that process large amounts of raw data, such as crawled documents, web request logs, etc., to compute various kinds of derived data, such as inverted indices, various representations of the graph structure of web documents, summaries of the number of pages crawled per host, the set of most frequent queries in a given day, etc. Most such computations are conceptually straightforward. However, the input data is usually large and the computations have to be distributed across hundreds or thousands of machines in order to finish in a reasonable amount of time. The issues of how to parallelize the computation, distribute the data, and handle failures conspire to obscure the original simple computation with large amounts of complex code to deal with these issues.

As a reaction to this complexity, we designed a new abstraction that allows us to express the simple computations we were trying to perform but hides the messy details of parallelization, fault-tolerance, data distribution and load balancing in a library. Our abstraction is inspired by the *map* and *reduce* primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a *map* operation to each logical "record" in our input in order to compute a set of intermediate key/value pairs, and then applying a *reduce* operation to all the values that shared the same key, in order to combine the derived data appropriately. Our use of a functional model with user-specified map and reduce operations allows us to parallelize large computations easily and to use re-execution as the primary mechanism for fault tolerance.

The major contributions of this work are a simple and powerful interface that enables automatic parallelization and distribution of large-scale computations, combined with an implementation of this interface that achieves high performance on large clusters of commodity PCs.

Section 2 describes the basic programming model and gives several examples. Section 3 describes an implementation of the MapReduce interface tailored towards our cluster-based computing environment. Section 4 describes several refinements of the programming model that we have found useful. Section 5 has performance measurements of our implementation for a variety of tasks. Section 6 explores the use of MapReduce within Google including our experiences in using it as the basis

To appear in OSDI 2004

1

Bigtable: A Distributed Storage System for Structured Data

Bigtable: A Distributed Storage System for Structured Data

Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber

{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com

Google, Inc.

Abstract

Bigtable is a distributed storage system for managing structured data that is designed to scale to a very large size: petabytes of data across thousands of commodity servers. Many projects at Google store data in Bigtable, including web indexing, Google Earth, and Google Finance. These applications place very different demands on Bigtable, both in terms of data size (from URLs to web pages to satellite imagery) and latency requirements (from backend bulk processing to real-time data serving). Despite these varied demands, Bigtable has successfully provided a flexible, high-performance solution for all of these Google products. In this paper we describe the simple data model provided by Bigtable, which gives clients dynamic control over data layout and format, and we describe the design and implementation of Bigtable.

1 Introduction

Over the last two and a half years we have designed, implemented, and deployed a distributed storage system for managing structured data at Google called Bigtable. Bigtable is designed to reliably scale to petabytes of data and thousands of machines. Bigtable has achieved several goals: wide applicability, scalability, high performance, and high availability. Bigtable is used by more than sixty Google products and projects, including Google Analytics, Google Finance, Orkut, Personalized Search, Writely, and Google Earth. These products use Bigtable for a variety of demanding workloads, which range from throughput-oriented batch-processing jobs to latency-sensitive serving of data to end users. The Bigtable clusters used by these products span a wide range of configurations, from a handful to thousands of servers, and store up to several hundred terabytes of data.

In many ways, Bigtable resembles a database: it shares many implementation strategies with databases. Parallel databases [14] and main-memory databases [13] have achieved scalability and high performance, but Bigtable provides a different interface than such systems. Bigtable does not support a full relational data model; instead, it provides clients with a simple data model that supports dynamic control over data layout and format, and allows clients to reason about the locality properties of the data represented in the underlying storage. Data is indexed using row and column names that can be arbitrary strings. Bigtable also treats data as uninterpreted strings, although clients often serialize various forms of structured and semi-structured data into these strings. Clients can control the locality of their data through careful choices in their schemas. Finally, Bigtable schema parameters let clients dynamically control whether to serve data out of memory or from disk.

Section 2 describes the data model in more detail, and Section 3 provides an overview of the client API. Section 4 briefly describes the underlying Google infrastructure on which Bigtable depends. Section 5 describes the fundamentals of the Bigtable implementation, and Section 6 describes some of the refinements that we made to improve Bigtable's performance. Section 7 provides measurements of Bigtable's performance. We describe several examples of how Bigtable is used at Google in Section 8, and discuss some lessons we learned in designing and supporting Bigtable in Section 9. Finally, Section 10 describes related work, and Section 11 presents our conclusions.

2 Data Model

A Bigtable is a sparse, distributed, persistent multi-dimensional sorted map. The map is indexed by a row key, column key, and a timestamp; each value in the map is an uninterpreted array of bytes.

(row:string, column:string, time:int64) → string

To appear in OSDI 2006

1

Dynamo: Amazon's Highly Available Key-value Store

Dynamo: Amazon's Highly Available Key-value Store

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati,
Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall
and Werner Vogels

Amazon.com

ABSTRACT

Reliability at massive scale is one of the biggest challenges we face at Amazon.com, one of the largest e-commerce operations in the world; even the slightest outage has significant financial consequences and impacts customer trust. The Amazon.com platform, which provides services for many web sites worldwide, is implemented on top of an infrastructure of tens of thousands of servers and network components located in many datacenters around the world. At this scale, small and large components fail continuously and the way persistent state is managed in the face of these failures drives the reliability and scalability of the software systems.

This paper presents the design and implementation of Dynamo, a highly available key-value storage system that some of Amazon's core services use to provide an "always-on" experience. To achieve this level of availability, Dynamo sacrifices consistency under certain failure scenarios. It makes extensive use of object versioning and application-assisted conflict resolution in a manner that provides a novel interface for developers to use.

Categories and Subject Descriptors

D.4.2 [**Operating Systems**]: Storage Management; D.4.5 [**Operating Systems**]: Reliability; D.4.2 [**Operating Systems**]: Performance;

General Terms

Algorithms, Management, Measurement, Performance, Design, Reliability.

1. INTRODUCTION

Amazon runs a world-wide e-commerce platform that serves tens of millions customers at peak times using tens of thousands of servers located in many data centers around the world. There are strict operational requirements on Amazon's platform in terms of performance, reliability and efficiency, and to support continuous growth the platform needs to be highly scalable. Reliability is one of the most important requirements because even the slightest outage has significant financial consequences and impacts customer trust. In addition, to support continuous growth, the platform needs to be highly scalable.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SOSP'07, October 14-17, 2007, Stevenson, Washington, USA.
Copyright 2007 ACM 978-1-59593-591-5/07/0010...$5.00.

One of the lessons our organization has learned from operating Amazon's platform is that the reliability and scalability of a system is dependent on how its application state is managed. Amazon uses a highly decentralized, loosely coupled, service oriented architecture consisting of hundreds of services. In this environment there is a particular need for storage technologies that are always available. For example, customers should be able to view and add items to their shopping cart even if disks are failing, network routes are flapping, or data centers are being destroyed by tornados. Therefore, the service responsible for managing shopping carts requires that it can always write to and read from its data store, and that its data needs to be available across multiple data centers.

Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time. As such Amazon's software systems need to be constructed in a manner that treats failure handling as the normal case without impacting availability or performance.

To meet the reliability and scaling needs, Amazon has developed a number of storage technologies, of which the Amazon Simple Storage Service (also available outside of Amazon and known as Amazon S3), is probably the best known. This paper presents the design and implementation of Dynamo, another highly available and scalable distributed data store built for Amazon's platform. Dynamo is used to manage the state of services that have very high reliability requirements and need tight control over the tradeoffs between availability, consistency, cost-effectiveness and performance. Amazon's platform has a very diverse set of applications with different storage requirements. A select set of applications requires a storage technology that is flexible enough to let application designers configure their data store appropriately based on these tradeoffs to achieve high availability and guaranteed performance in the most cost effective manner.

There are many services on Amazon's platform that only need primary-key access to a data store. For many services, such as those that provide best seller lists, shopping carts, customer preferences, session management, sales rank, and product catalog, the common pattern of using a relational database would lead to inefficiencies and limit scale and availability. Dynamo provides a simple primary-key only interface to meet the requirements of these applications.

Dynamo uses a synthesis of well known techniques to achieve scalability and availability: Data is partitioned and replicated using consistent hashing [10], and consistency is facilitated by object versioning [12]. The consistency among replicas during updates is maintained by a quorum-like technique and a decentralized replica synchronization protocol. Dynamo employs

205

ACID

ACID! The King is Dead! Long Live the King!

atomic

consistent

isolated

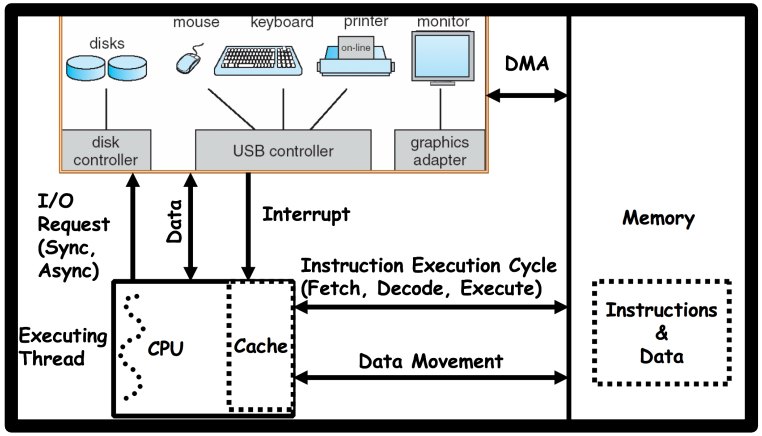
durable

In order to keep CPU and I/O Busy…

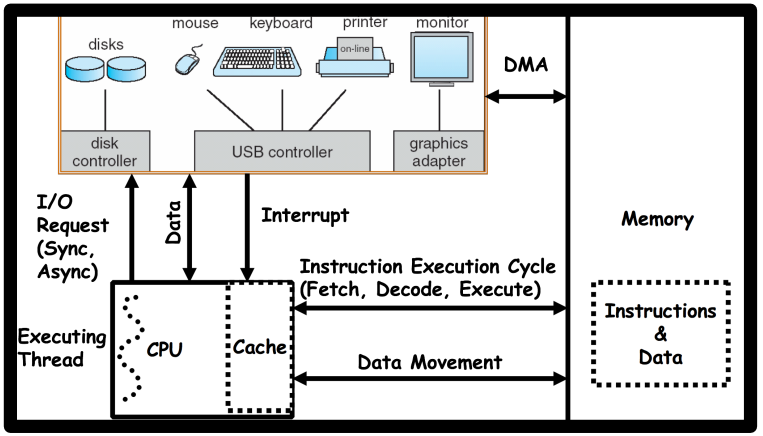
- We invented the concept of "Processes".
- What are these?

Process Abstraction…

Virtual Machine & CPU 1

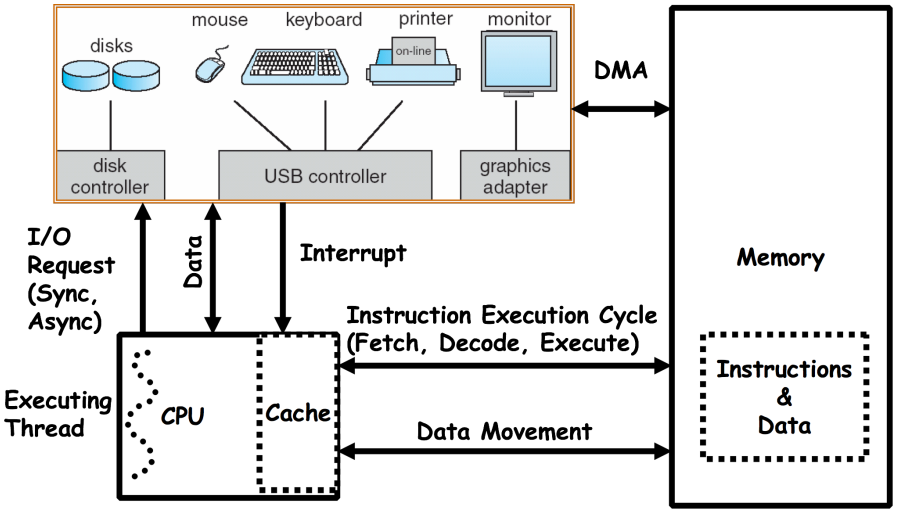


Virtual Machine & CPU 2



The Real Machine & CPU

The Context Switch!

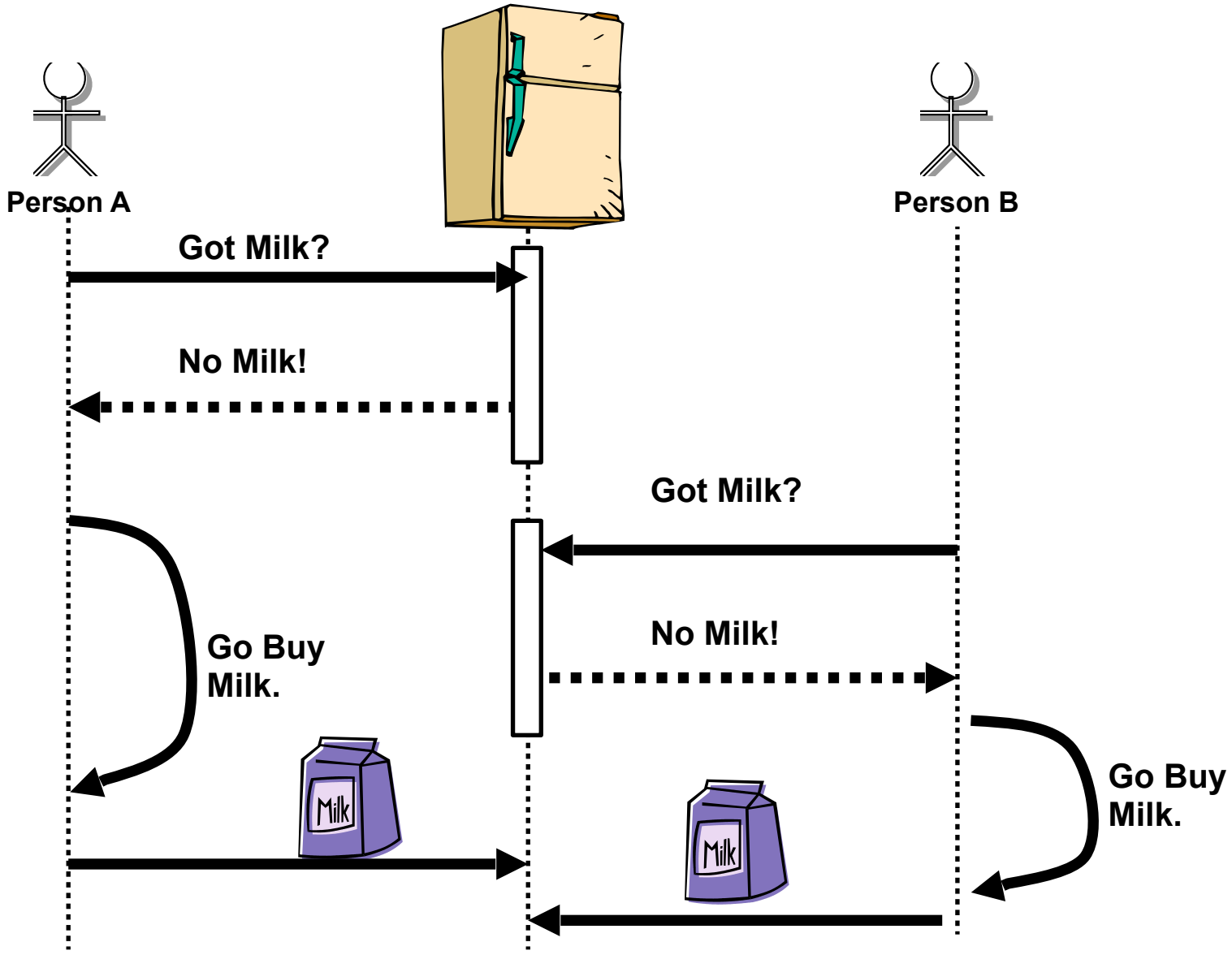


Illusion of Virtual Machine not perfect!

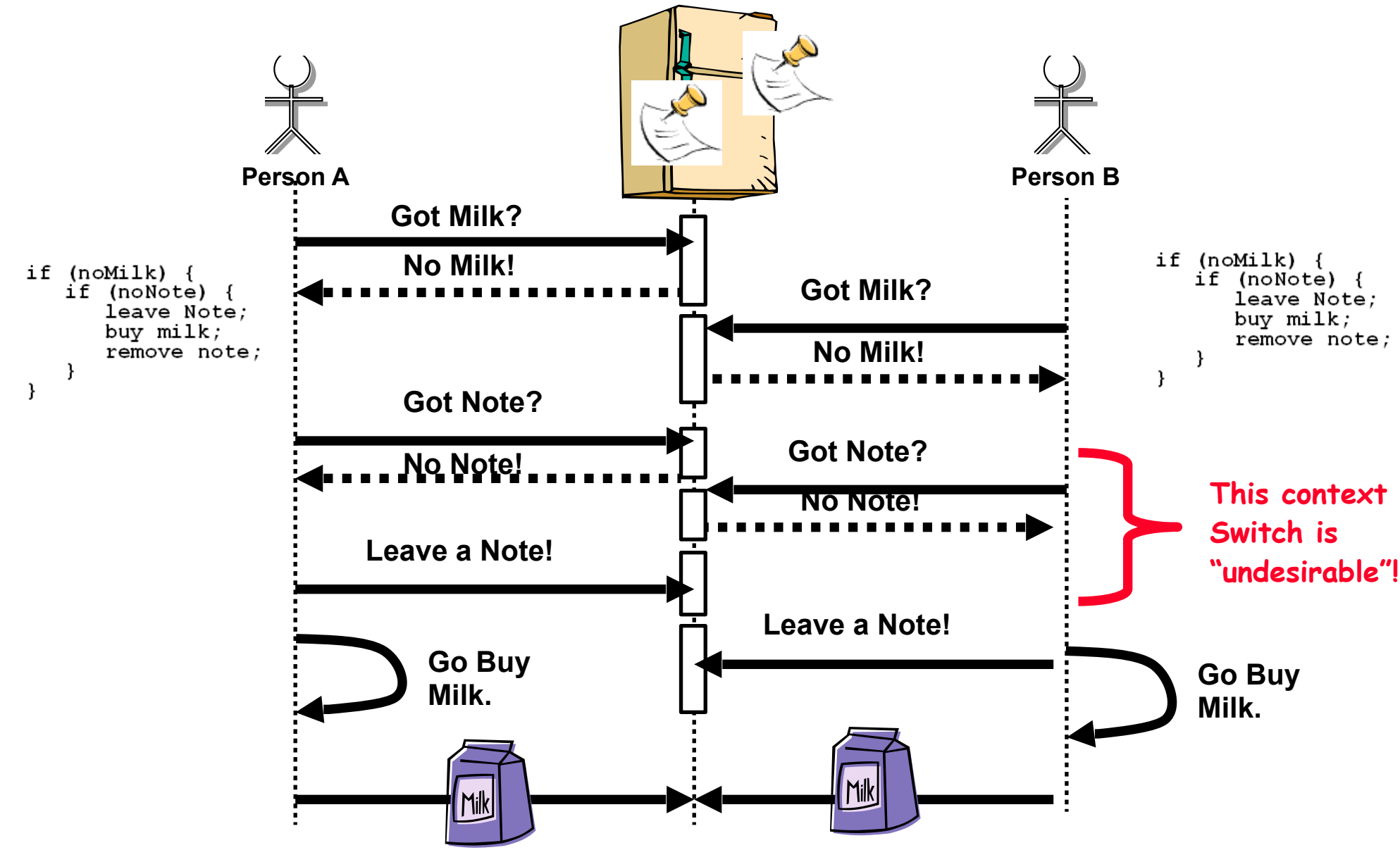
We have to deal with concurrent access to shared memory.

AKA => "Race Condition"

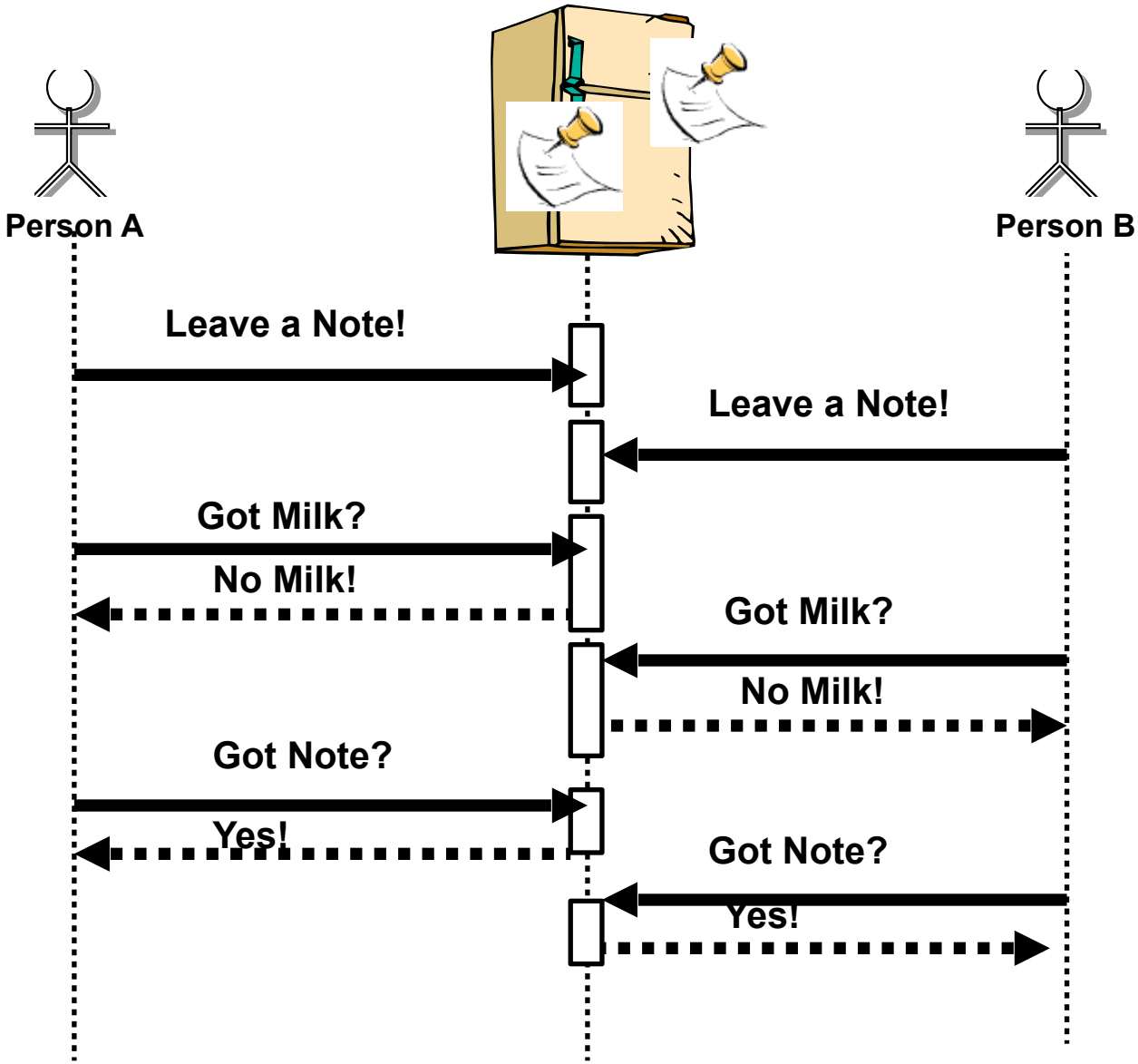
Too Much Milk! – The Problem.



Okay, we can solve this… just leave a note!



How about leaving a note first?



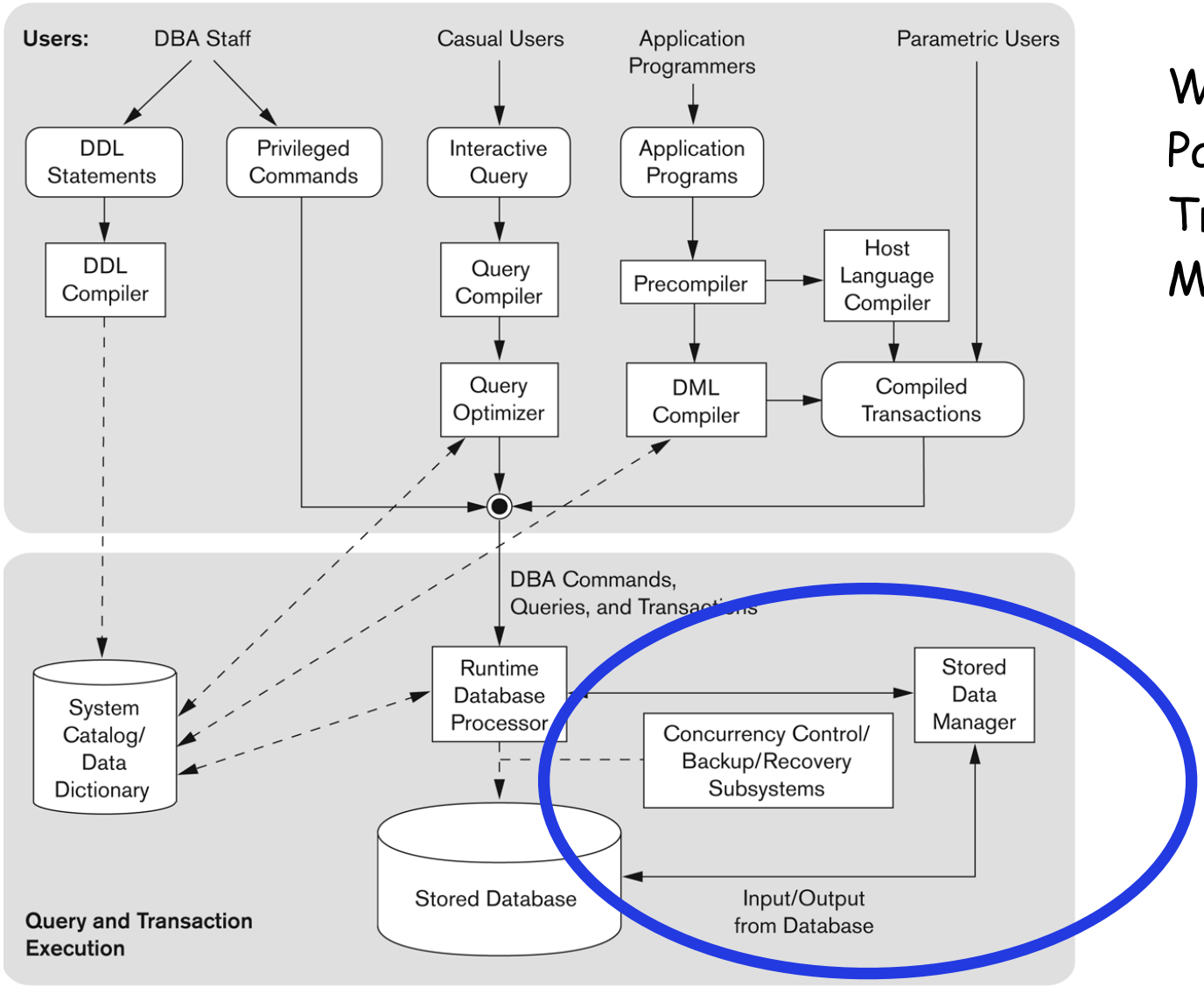

```
leave Note;
if (noMilk) {
    if (noNote) {
        leave Note;
        buy milk;
    }
}
remove note;
```

Solutions to concurrent access/updates are founded on…

The basic idea that we need "atomic" operations which can not be interrupted (e.g. context switched in the middle).

Q: Are all H/W instructions atomic?

Major Subsystems of an RDBMS



Which of these Participate in Transaction Management?

Subsystems Of a DB involved In managing Transactions.

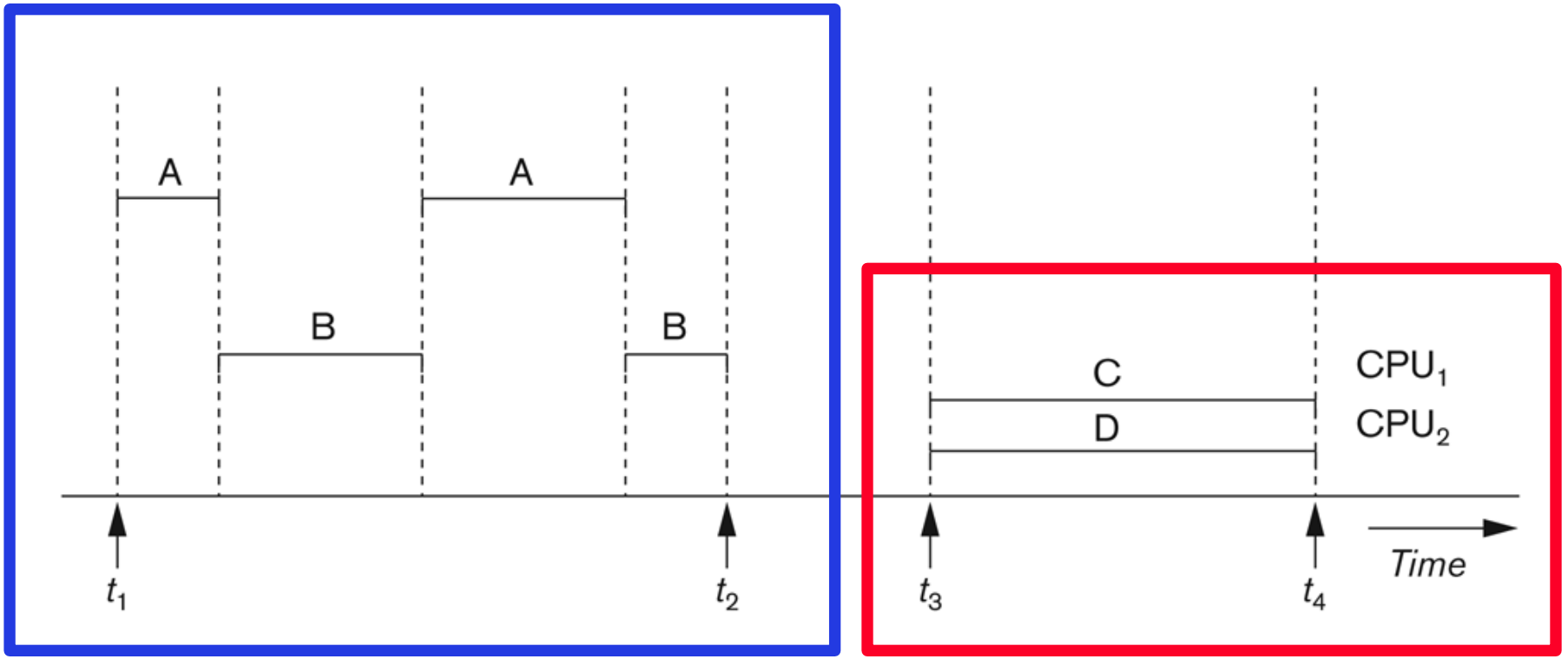
Figure 2.3
Component modules of a DBMS and their interactions.

What can go wrong?

Transactions Executing Concurrently

- **Interleaved processing**
  - Concurrent execution of processes is interleaved in a single CPU
- **Parallel processing**
  - Processes are concurrently executed in multiple CPUs

Text



Sally's Program

- Sally executes the following two SQL statements, which we call (min) and (max), to help remember what they do.

(max)
```
SELECT MAX(price) FROM Sells
  WHERE bar = 'Joe''s Bar';
```

(min)
```
SELECT MIN(price) FROM Sells
  WHERE bar = 'Joe''s Bar';
```

Joe's Program

- At about the same time, Joe executes the following steps, which have the mnemonic names (del) and (ins).

```
(del)    DELETE FROM Sells
             WHERE bar = 'Joe''s Bar';

(ins)    INSERT INTO Sells
             VALUES('Joe''s Bar', 'Heineken',
             3.50);
```

Interleaving of Statements

- Although (max) must come before (min) and (del) must come before (ins)

- There are no other constraints on the order of these statements (1).

Example: Strange Interleaving

- Suppose the steps execute in the order

	(max)	(del)	(ins)	(min)
	2.50, 3.00	2.50, 3.00		3.50
Joe's Prices:	**(max)**	(del)	(ins)	**(min)**
Statement:	**3.00**			**3.50**
Result:				

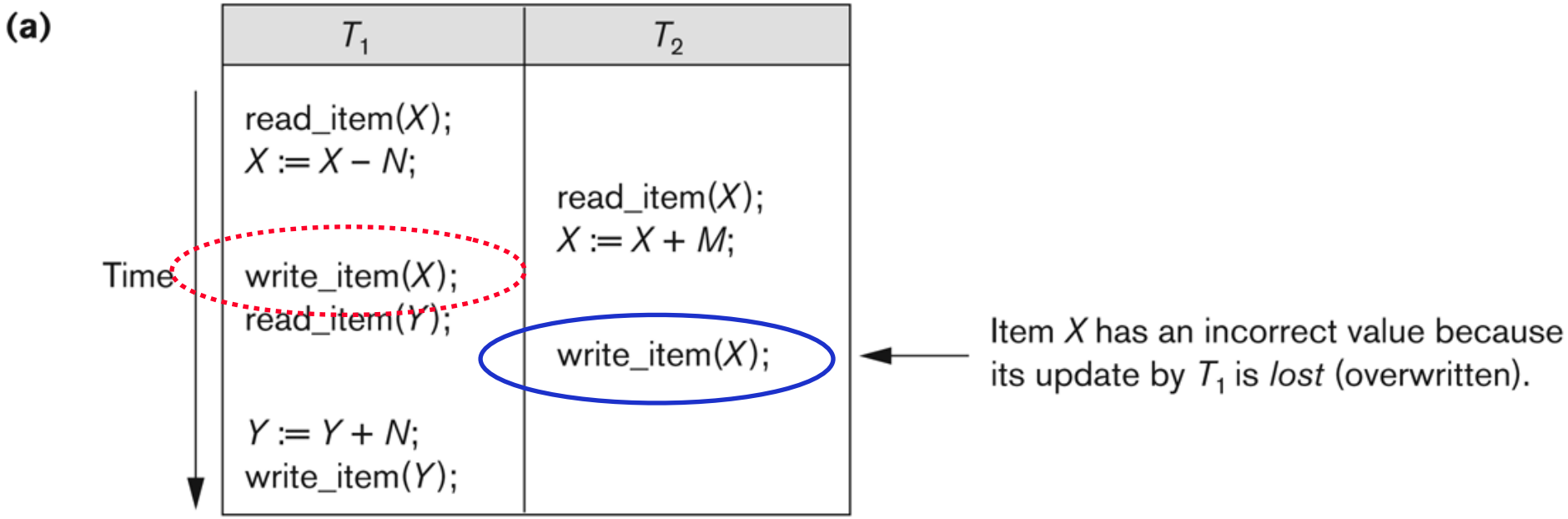
- Sally sees MAX < MIN!

Three Problems with Concurrent Execution

- The Lost Update Problem
  - Concurrent writes
- The Temporary Update Problem
  - Dirty read
- The Incorrect Summary Problem
  - Computing results based on inconsistent reads

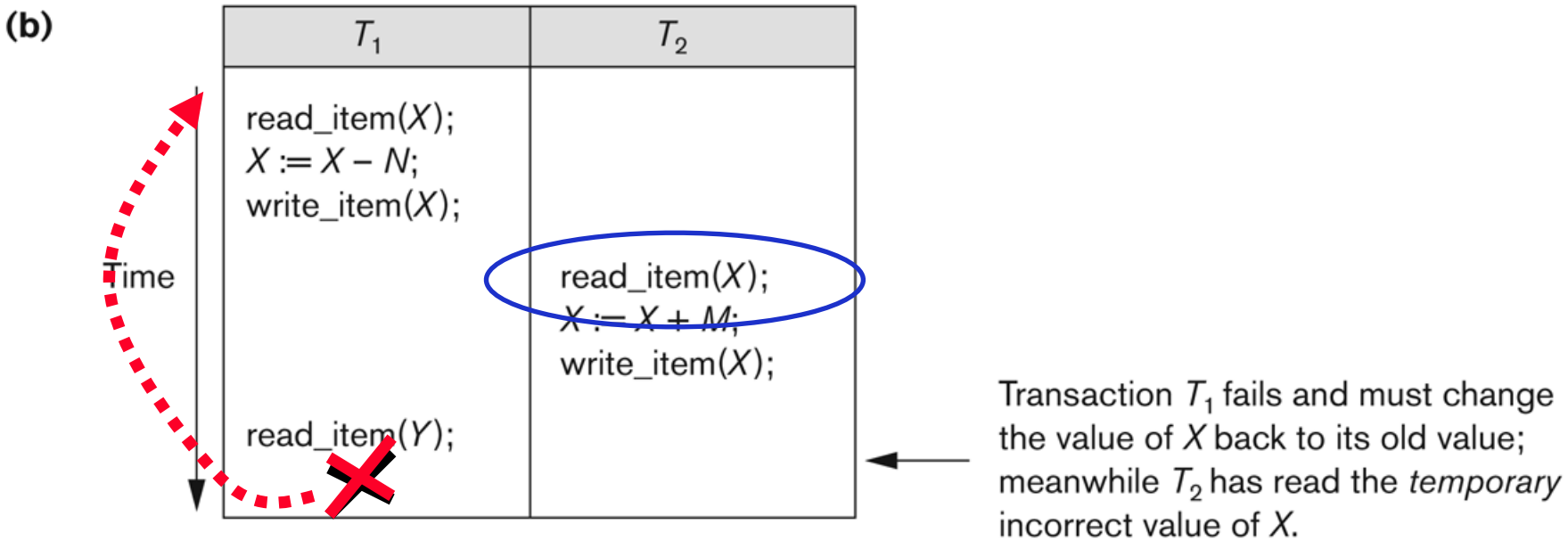
The lost update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.



The temporary update problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

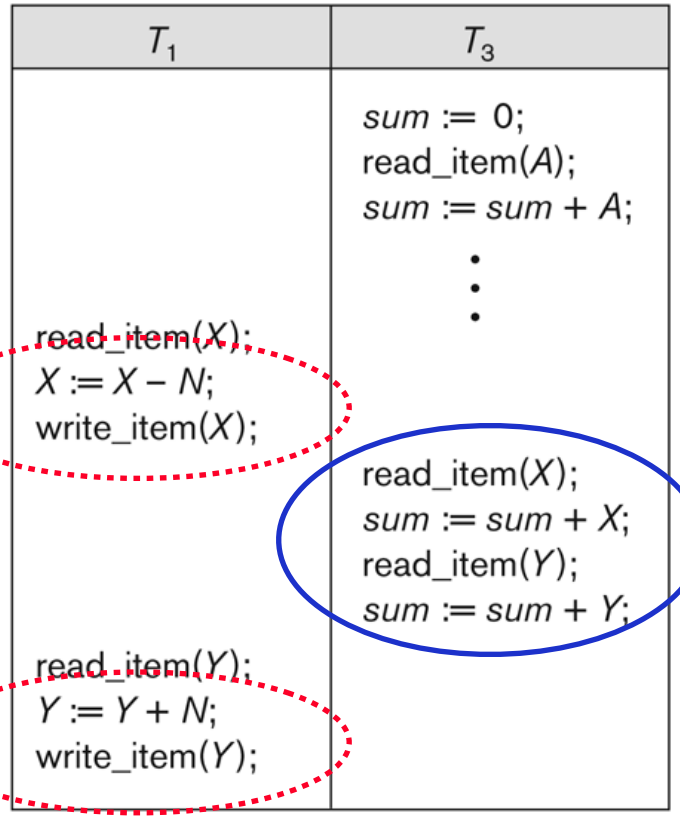


The incorrect summary problem.

Figure 17.3
Some problems that occur when concurrent execution is uncontrolled. (a) The lost update problem. (b) The temporary update problem. (c) The incorrect summary problem.

A = 1
X = 10
Y = 10

(c)



X = X - 5 (5)

Y = 10 + 5 (15)

A = 1
X = 5
Y = 15

SUM = 1

SUM = 1 + 5 + 10 (16)

T_3 reads X after N is subtracted and reads Y before N is added; a wrong summary is the result (off by N).

SUM = 16

ACID

atomic

consistent

isolated

durable

ACID Properties

- **Atomicity**
  - A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency preservation**
  - A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**
  - A transaction should not make its updates visible to other transactions until it is committed; this property, when enforced strictly, solves the temporary update problem and makes cascading rollbacks of transactions unnecessary (see Chapter 21).
- **Durability or permanency**
  - Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

Notation for Schedules

(a)

T_1

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);

(b)

T_2

read_item(X);
$X := X + M$;
write_item(X);

Operations:

r = Read
w = Write
c = Commit
a = Abort

(a) T1 = $r_1(X);w_1(X);r_1(Y);w_1(Y)$

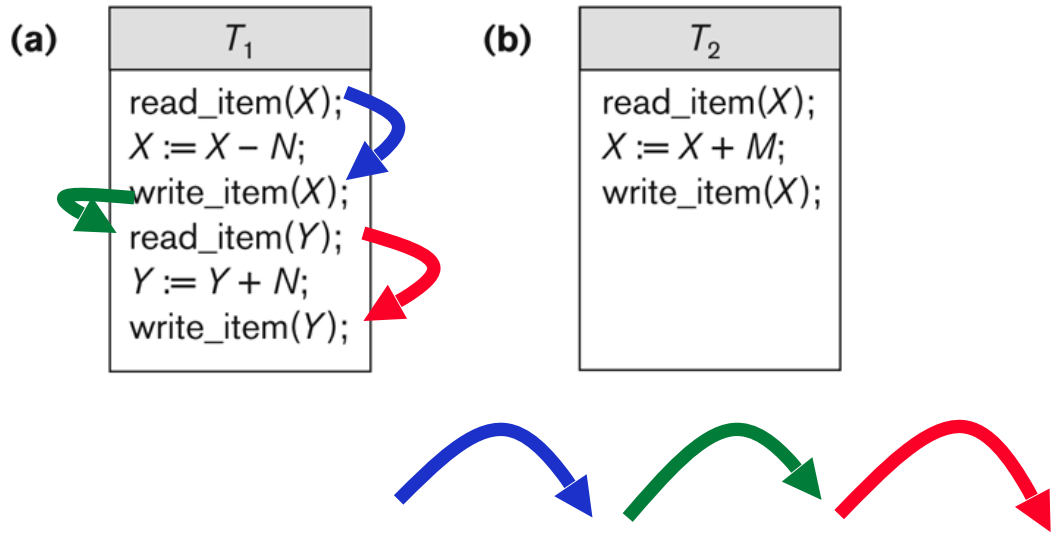
Operation

Transaction ID

Database Item

(b) T2 = $r_2(X);w_2(X)$

Notation for Schedules (cont.)



T_1	T_2
read_item(X); X := X − N;	
	read_item(X); X := X + M;
write_item(X); read_item(Y);	
	write_item(X);
Y := Y + N; write_item(Y);	

(a) T1 = $r_1(X);w_1(X);r_1(Y);w_1(Y)$

(b) T2 = $r_2(X);w_2(X)$

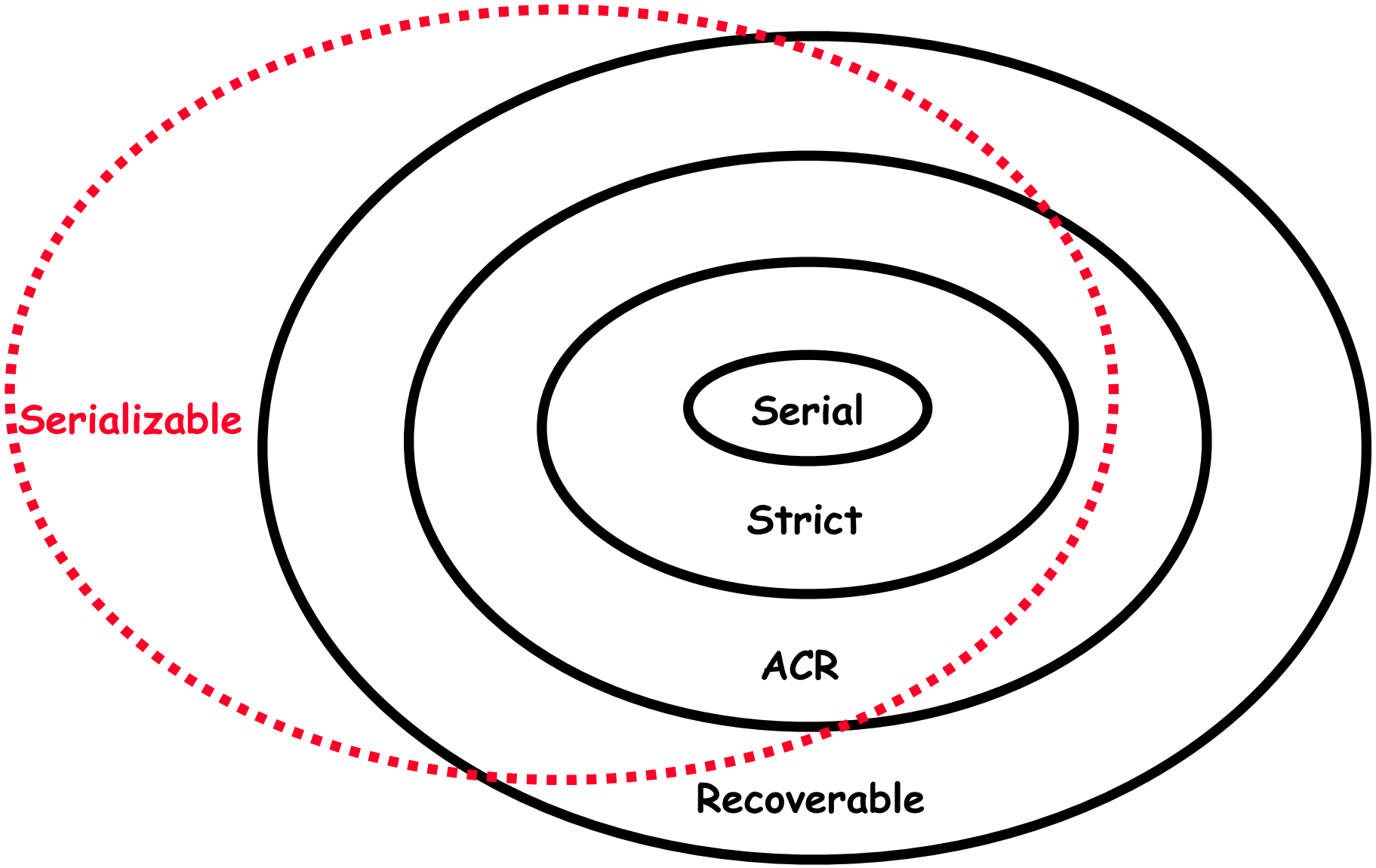
S = T1;T2 = $r_1(X);w_1(X);r_1(Y);w_1(Y);\ r_2(X);w_2(X)$

S' = $r_1(X);r_2(X);w_1(X);r_1(Y);w_2(X);w_1(Y)$

S is a serial schedule

S' is a schedule where operations are interleaved

Classifying Schedules



ACR = Avoids Cascading Rollbacks

Transaction Support in SQL2 (example)

- Sample SQL transaction:

```
EXEC SQL whenever sqlerror go to UNDO;

EXEC SQL SET TRANSACTION
          READ WRITE
          DIAGNOSTICS SIZE 5
          ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT
         INTO EMPLOYEE (FNAME, LNAME, SSN, DNO, SALARY)
         VALUES ('Robert','Smith','991004321',2,35000);
EXEC SQL UPDATE EMPLOYEE
         SET SALARY = SALARY * 1.1
         WHERE DNO = 2;
EXEC SQL COMMIT;
         GOTO  THE_END;

UNDO: EXEC SQL ROLLBACK;

THE_END:  ...
```

Possible violation of serializabilty

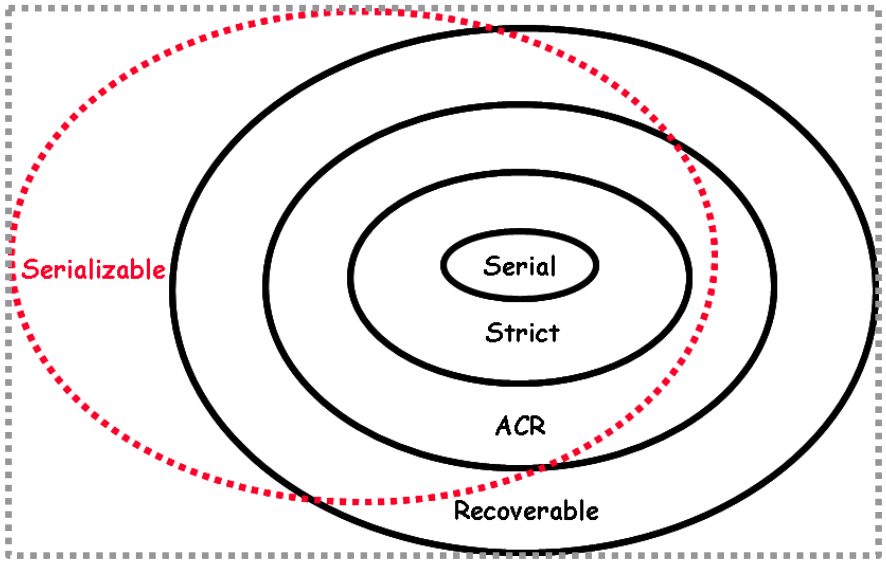
	Type of Violation		
Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommited	Yes	Yes	Yes
Read Commited	No	Yes	Yes
Repeatable Read	No	No	Yes
Serializable	No	No	No

Not Recoverable:

Requires Casacading Rollbacks!

Strict Locking:

Hold locks until abort or commit. Row-Level Lock?



ACR:

Only Read Committed Data

Serial/-izable:

Need to be equiv. to some serial schedule. Locking Policy? Table Level Lock?

Oracle Example

Transactions

SET TRANSACTION

Purpose

Use the SET TRANSACTION statement to establish the current transaction as read-only
or read/wri

The operati
transaction,

READ WRITE

Specify `READ WRITE` to establish the current transaction as a read/write transaction. This clause establishes **statement-level read consistency**, which is the default.

tion language (DDL)

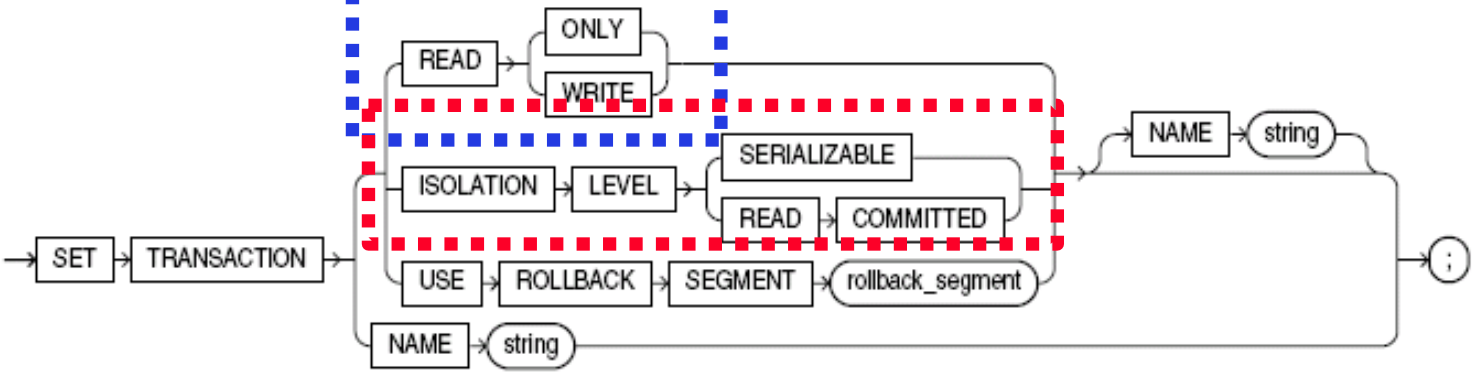
nt transaction as a read-only transaction.
ad consistency.

ee only changes that were committed
nsactions are useful for reports that run
while other users update these same

ISOLATION LEVEL Clause

Use the `ISOLATION LEVEL` clause to specify how transactions containing database modifications are handled.

- The `SERIALIZABLE` setting specifies serializable transaction isolation mode as defined in the SQL92 standard. If a serializable transaction contains data manipulation language (DML) that attempts to update any resource that may have been updated in a transaction uncommitted at the start of the serializable transaction, then the DML statement fails.
- The `READ COMMITTED` setting is the default Oracle Database transaction behavior. If the transaction contains DML that requires row locks held by another transaction, then the DML statement waits until the row locks are released.



Transaction Example
(What's the Outcome for Test 1 and Test 2?)

Test:

```
Test 1:  Default isolation levels for both.
         S = T1(1);T2(1);T2(2);T1(2);T2(3);T1(3)
Test 2:  T1 at serializable. T2 at default.
         S=T1(1);T2(1);T2(2);T2(3);T1(2);T1(3)
```

Setup:

```
drop table bank_account ;
create table bank_account
( id number, type char(3), balance number(7,2)) ;
insert into bank_account values ( 1, 'CHK', 1000) ;
insert into bank_account values ( 2, 'SAV', 1000) ;
```

T1 (Transfer from Checking to Savings):

```
1. select * from bank_account ;
2. update bank_account set balance = balance - 500 where id = 1 ;
3. update bank_account set balance = balance + 500 where id = 2 ;
```

T2 (Transfer from Savings to Checking):

```
1. select * from bank_account ;
2. update bank_account set balance = balance - 500 where id = 2 ;
3. update bank_account set balance = balance + 500 where id = 1 ;
```

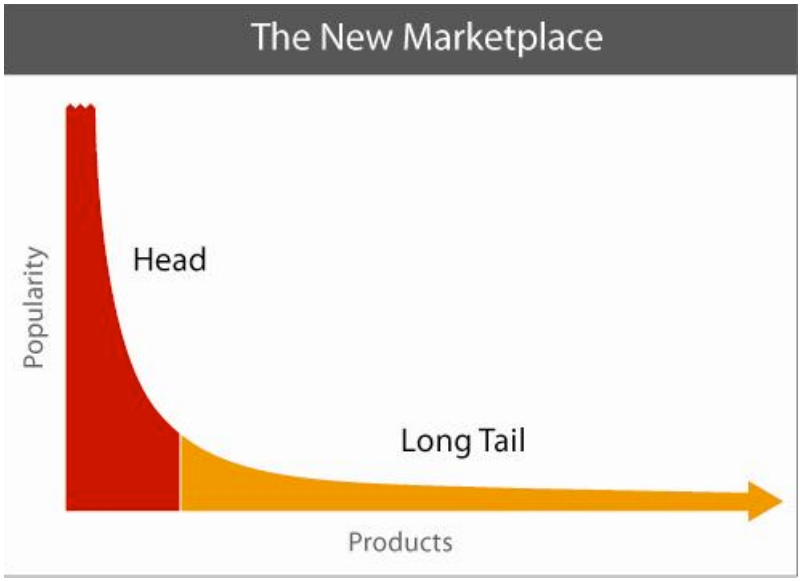
Isolation Levels:

```
set transaction isolation level read committed ; // default
set transaction isolation level serializable ;
```

NoSQL

Cloud Service Models

The Long Tail



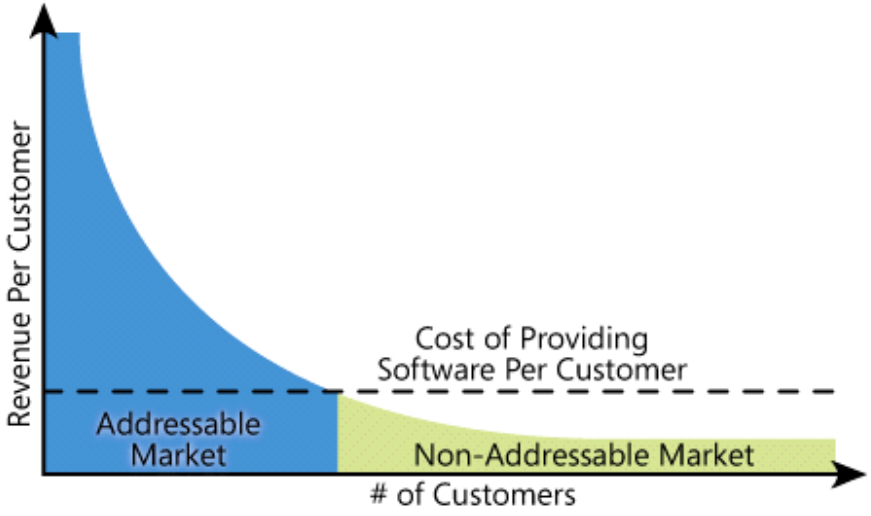


Figure 6. Market curve for LOB software vendors

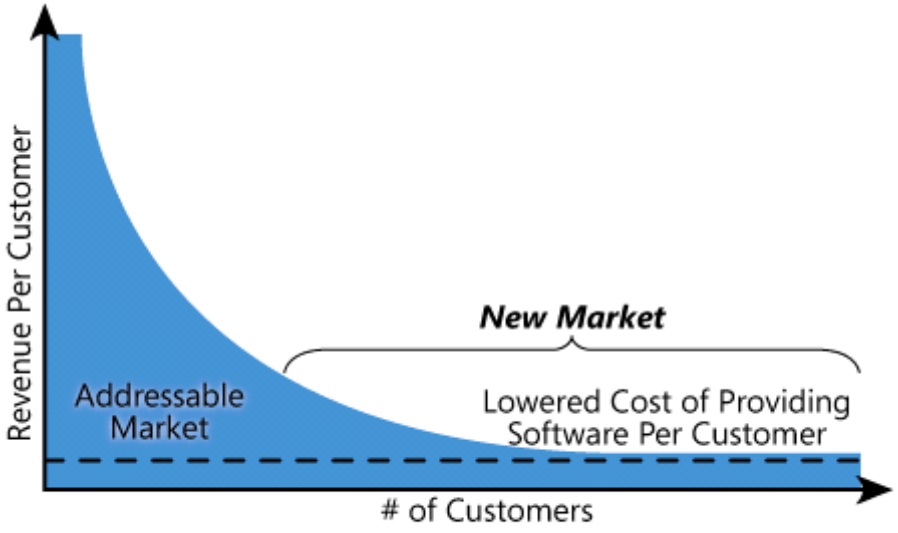


Figure 7. New market opened by lower cost of SaaS

The Cloud: *"a brave new world"*

	Software	Platform	Infrastructure
Head	Manual Customization	JVM (or .Net)	Dedicated Hardware
Long Tail	Mass Customization	Multi-Paradigm	Virtual Machine
Challenges	Single Instance	Single Runtime	Scaling Data

Polymorphic Application

- **Current Approach**

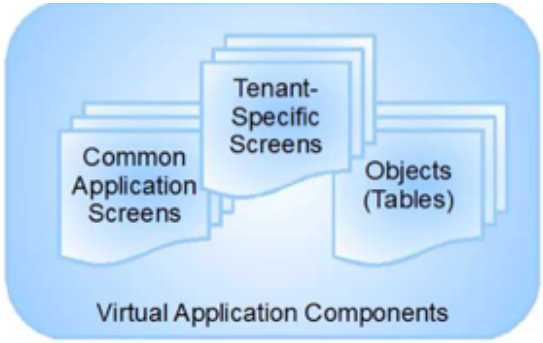
  - MetaData Driven

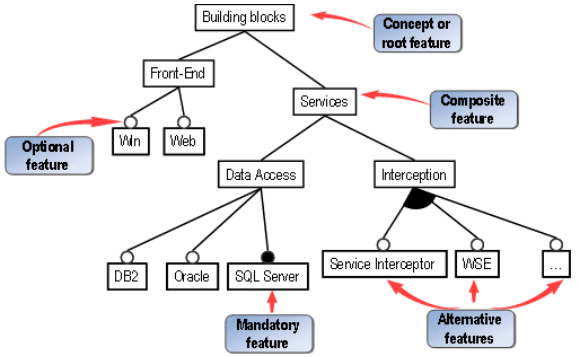
  1. Dedicated Database
  2. Shared DB, Fixed Extension Set
  3. Shared DB, Custom Extensions

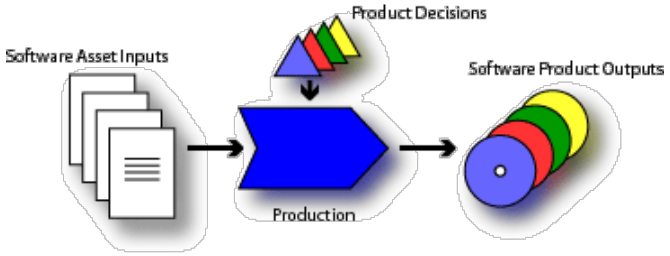
- **Emerging Approach:**

  - ByteCode Injection

  1. Beyond Objects
  2. Aspect-Oriented Weaving
  3. Product-Line Engineering







Polyglot Programming

- **Current Approach**

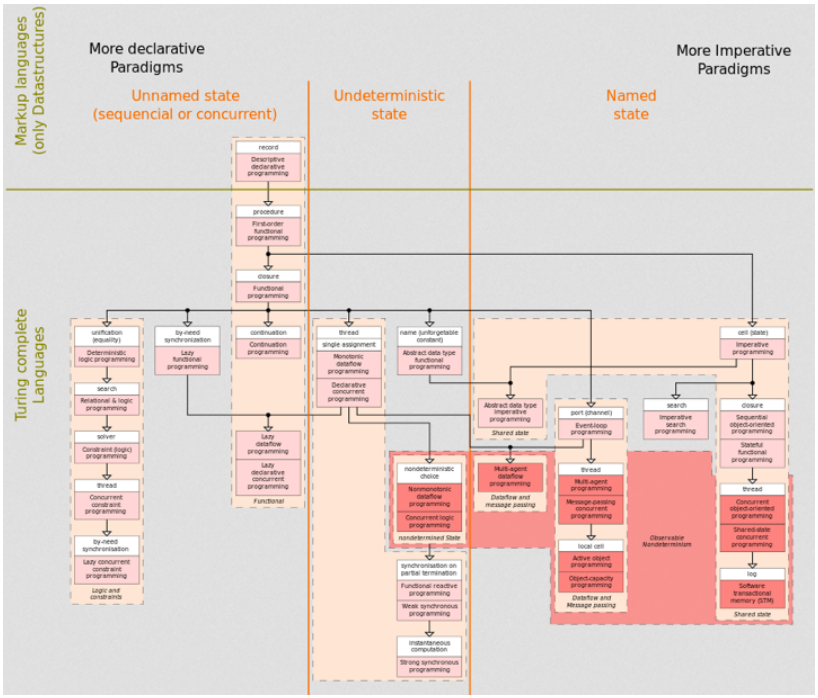
  - *Single Paradigm Design*

  1. Object-Oriented Analysis
  2. Object-Oriented Design
  3. Object-Oriented Programming

- **Emerging Approach:**

  - *Multi-Paradigm Design*

  1. Beyond Objects
  2. Multiple Languages (GSL + DSL)
  3. OOP + Functional + AOP



http://polyglotprogramming.com/

http://en.wikipedia.org/wiki/File:Programming_paradigms.svg

Polyglot Persistence

- **Current Approach**

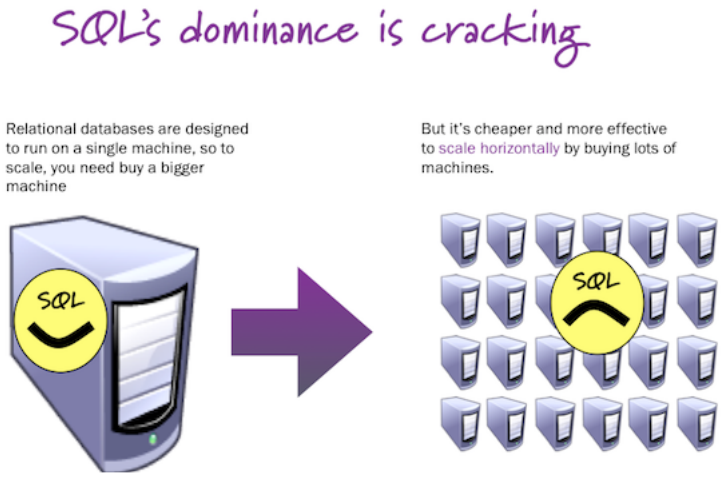
  - Relational is King!

  1. OLTP (Transactions)
  2. OLAP (Analytics)
  3. Centralized DB Server
  4. Scale-Up (add expensive hardware)

- **Emerging Approach:**

  - Not Just Relational

  1. SQL + NoSQL (Eventual Consistency)
  2. Hadoop + Columnar DB (Analytics)
  3. Distributed Data Stores
  4. Scale-Out (add cheap computers)



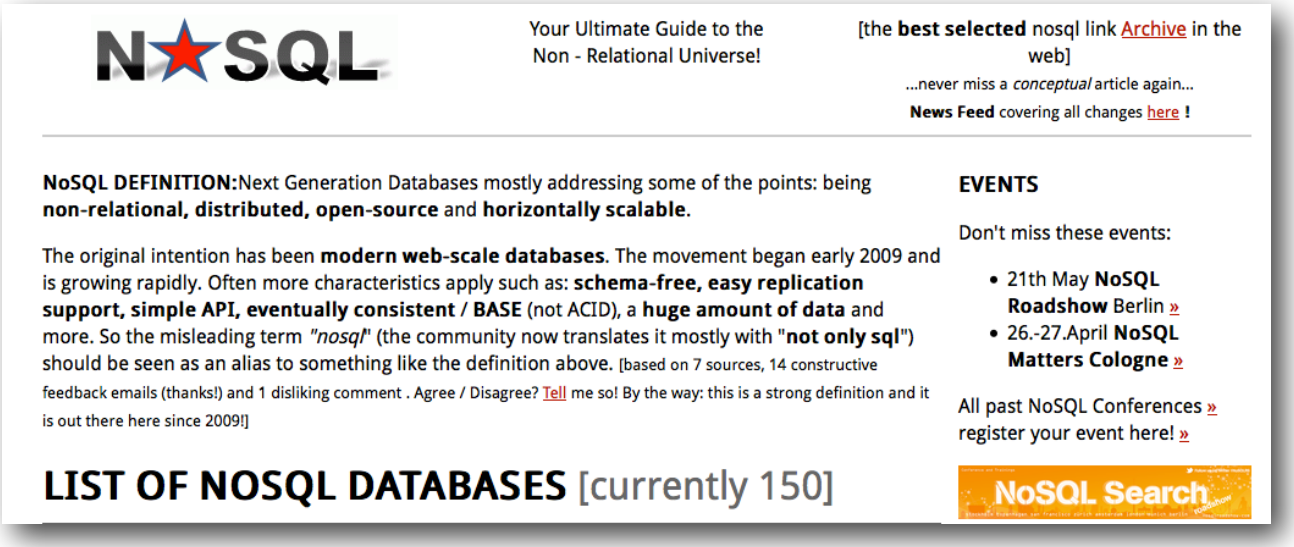
http://martinfowler.com/bliki/PolyglotPersistence.html

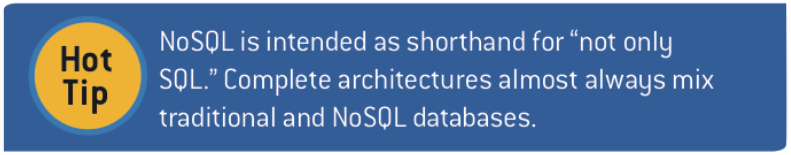
The term “NoSQL”

No SQL Language
(*1998 Definition*)

The term *“NoSQL”* was originally used in 1998 to describe a new lightweight open-source relational DBMS that did not expose an SQL interface. The name was then reprised and used 10 years later by Eric Evans (of Rackspace) and Johan Oskarsson (of Last.fm) for the NoSQL meetup/conference focusing on the topic of “*open-source, distributed, non relational databases*“.

Not Just SQL
(2008 Definition)
Evans & Oskarsson



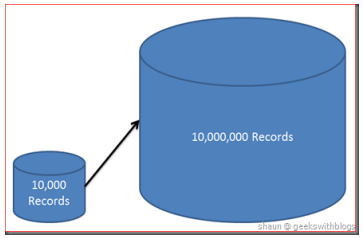


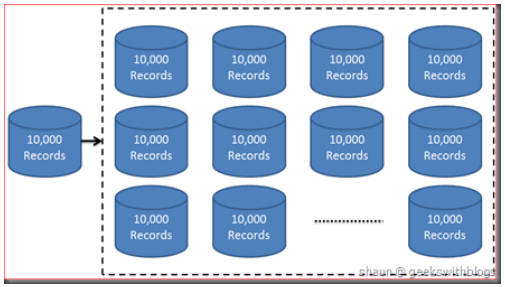
http://blog.rizzif.com/2011/08/31/intro-to-nosql/

http://nosql-database.org/

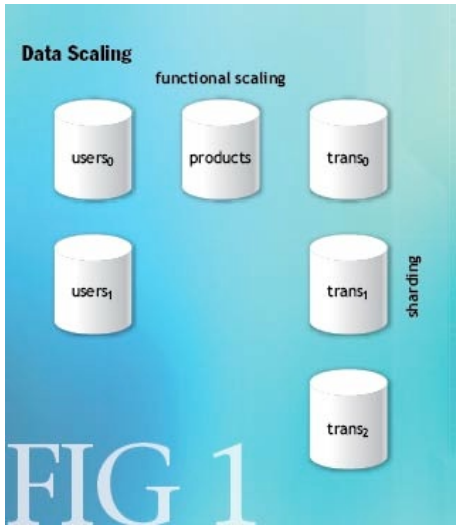
The "Scaling" Problem

- **Vertical Scaling** (*aka:* Move everything to a bigger, faster infrastructure) is limited by certain limits (there is a hard-cap upper bound which is the fastest machine available at the time, is expensive, suffers from vendor lock, and is overall inefficient since the resources have to be acquired ahead of the time when they will be needed).
- **Sharding** (*aka:* Horizontal Partitioning) faces its own limitations (the Sharding Principle is not always easy to determine, inter-shard communications are still necessary, rebalancing is difficult and expensive) and enjoys limited support from the traditional RDBMSes vendors.
- **Horizontal Scaling**, on the other hand, seems to be the best solution for scaling a system over time. However, it is a difficult approach to implement when using a traditional RDBMS.





Web applications have grown in popularity over the past decade. Whether you are building an application for end users or application developers (i.e., services), your hope is most likely that your application will find broad adoption—and ***with broad adoption will come transactional growth****. If your application relies upon persistence, then data storage will probably become your bottleneck.*



Functional Partitioning

Functional partitioning is important for achieving high degrees of scalability. Any good database architecture will decompose the schema into tables grouped by functionality. Users, products, transactions, and communication are examples of functional areas. Leveraging database concepts such as foreign keys is a common approach for maintaining consistency across these functional areas

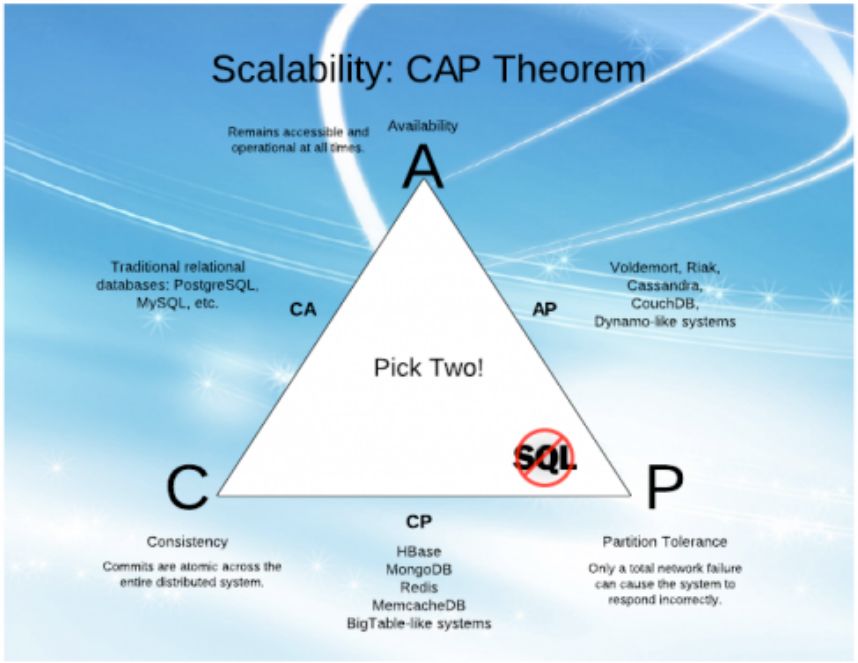
http://blog.rizzif.com/2011/08/31/intro-to-nosql/

http://queue.acm.org/detail.cfm?id=1394128

Brewer's CAP Theorem

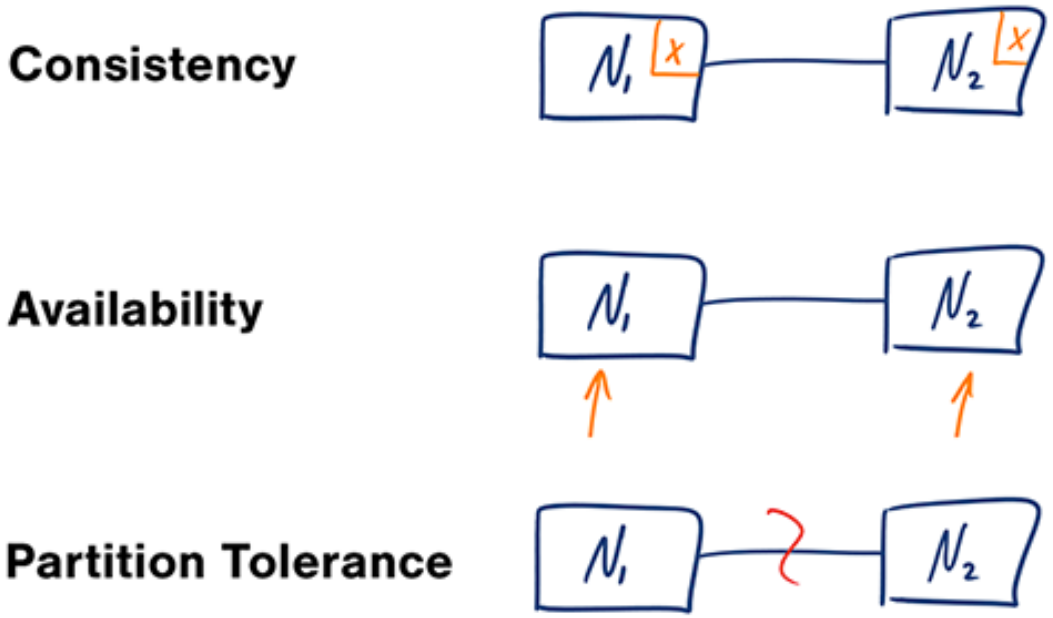
Eric Brewer, a professor at the University of California, Berkeley, and cofounder and chief scientist at Inktomi, made the conjecture that Web services cannot ensure all three of the following properties at once (signified by the acronym CAP):[2] Specifically, a Web application can support, at most, only two of these properties with any database design. Obviously, any horizontal scaling strategy is based on data partitioning; therefore, designers are forced to decide between consistency and availability.

- **Consistency** means that each client always has the same view of the data.
- **Availability** means that all clients can always read and write.
- **Partition tolerance** means that the system works well across physical network partitions.



http://blog.rizzif.com/2011/08/31/intro-to-nosql/

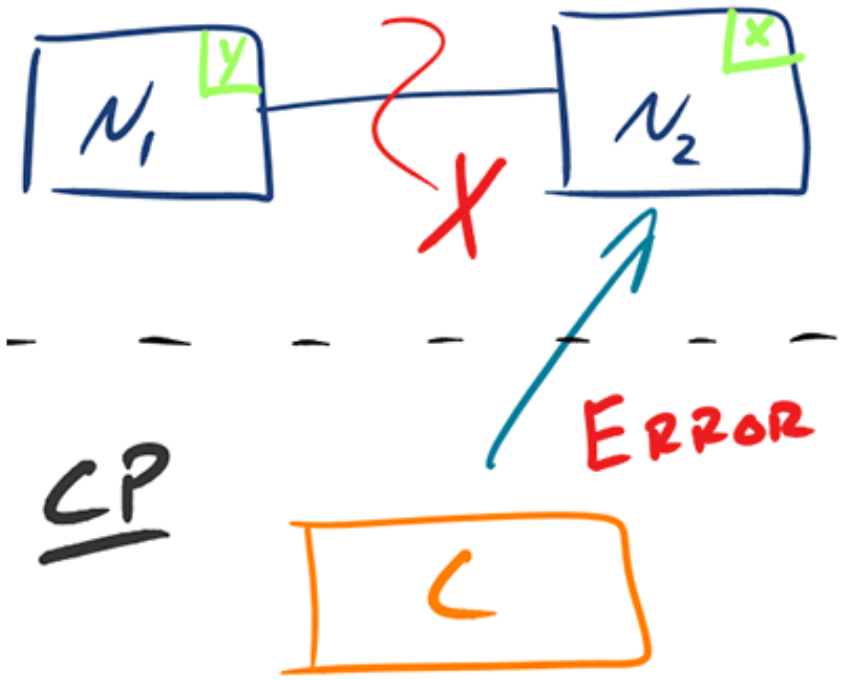
http://queue.acm.org/detail.cfm?id=1394128



- **Consistency** - A read is guaranteed to return the most recent write for a given client.
- **Availability** - A non-failing node will return a reasonable response within a reasonable amount of time (no error or timeout).
- **Partition Tolerance** - The system will continue to function when network partitions occur.

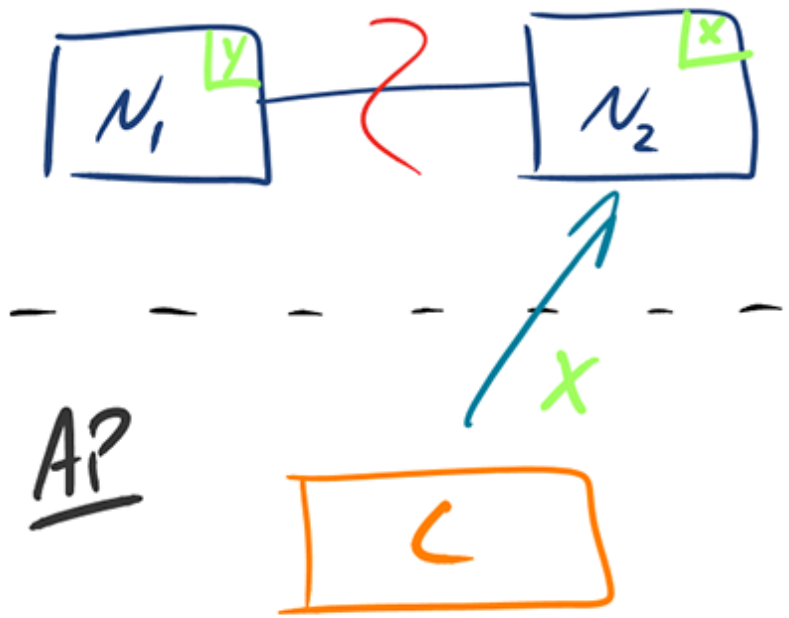
http://robertgreiner.com/2014/08/cap-theorem-revisited/

- **CP** - Consistency/Partition Tolerance - Wait for a response from the partitioned node which could result in a timeout error. The system can also choose to return an error, depending on the scenario you desire. Choose Consistency over Availability when your business requirements dictate atomic reads and writes.



http://robertgreiner.com/2014/08/cap-theorem-revisited/

- **AP** - Availability/Partition Tolerance - Return the most recent version of the data you have, which could be stale. This system state will also accept writes that can be processed later when the partition is resolved. Choose Availability over Consistency when your business requirements allow for some flexibility around when the data in the system synchronizes. Availability is also a compelling option when the system needs to continue to function in spite of external errors (shopping carts, etc.)



http://robertgreiner.com/2014/08/cap-theorem-revisited/

ACID vs. BASE

ACID

In the database world, ACID is an acronym that says, all database transactions should be:

- **atomic**: transaction is either succeed or entirely rolled back
- **consistent**: transaction never invalidates the database state
- **isolated**: transactions should not interfere with each other but done as if they are sequential
- **durable**: completed transactions persist, i.e. what you wrote is what you read

The ACID property is a strict requirement for database. It impairs the availability and performance. For example, ACID makes database making frequent locks on tables to guarantee strong consistency. The lock hurts I/O throughput and make the database not responsive until the lock is over.

BASE

BASE is described as the other end of a spectrum by Eric Brewer is his 2000 PODC keynote speech. It is:

- **basically available**: system looks to work and responsive at any time
- **soft-stat**e: no need to be consistent at all time
- **eventually consisten**t: but become consistent at some later time

BASE means that, consistency after every transaction is not required, as long as the database is eventually in a consistent state. Therefore, using stale data or providing approximate answers are tolerated. In essence, BASE is a best-effort design that gives up strong consistency for weak consistency.

http://adrian.idv.hk/doku.php/research/captheorem

ACID vs. BASE

The CAP theorem is given out by Eric Brewer in his PODC keynote speech and proved by Gilbert and Lynch in 2002. It says that it is impossible to achieve all three of:

- consistency: i.e., atomicity in ACID
- availability: i.e., responsiveness of the system
- partition tolerance: i.e., system works (amid partially) even with partial failure

Examples of systems applying CAP theorem:

- single-site database: Forfeits P for C and A
- distributed database with distributed locking: Forfeits A for C and P
- web caching and DNS: Forfeits C for A and P

Therefore, in dealing with the CAP theorem, the strategy in designing a distributed system are:

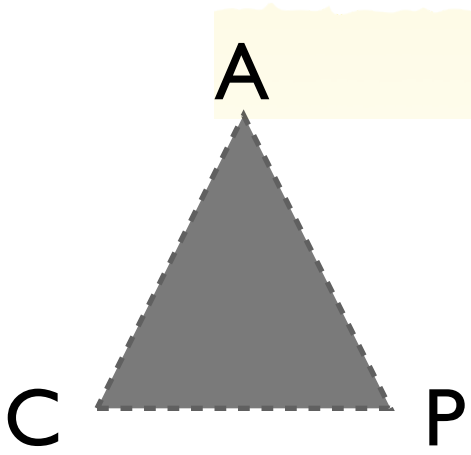
1. Drop partition tolerance:
    - put everything into one machine
    - imposes scaling limits
2. Drop availability
    - upon partition event (i.e. partial system failure), affected service simply wait
    - until data is consistent, system remain unavailable/irresponsive
3. Drop consistency
    - accept things will become eventually consistent

http://adrian.idv.hk/doku.php/research/captheorem

Examples of CAP Applied

Consistent, Available (CA) Systems have trouble with partitions and typically deal with it with replication. Examples of CA systems include:

- Traditional RDBMSs like Postgres, MySQL, etc (relational)
- Vertica (column-oriented)
- Aster Data (relational)
- Greenplum (relational)



Available, Partition-Tolerant (AP) Systems achieve "eventual consistency" through replication and verification. Examples of AP systems include:

- Dynamo (key-value)
- Voldemort (key-value)
- Tokyo Cabinet (key-value)
- KAI (key-value)
- Cassandra (column-oriented/tabular)
- CouchDB (document-oriented)
- SimpleDB (document-oriented)
- Riak (document-oriented)

Consistent, Partition-Tolerant (CP) Systems have trouble with availability while keeping data consistent across partitioned nodes. Examples of CP systems include:

- BigTable (column-oriented/tabular)
- Hypertable (column-oriented/tabular)
- HBase (column-oriented/tabular)
- MongoDB (document-oriented)
- Terrastore (document-oriented)
- Redis (key-value)
- Scalaris (key-value)
- MemcacheDB (key-value)
- Berkeley DB (key-value)

http://blog.nahurst.com/visual-guide-to-nosql-systems

Key Value stores

Examples: Tokyo Cabinet/Tyrant, Redis, Voldemort, Oracle BDB Typical applications: Content caching Strengths: Fast lookups Weaknesses: Stored data has no schema

Graph databases

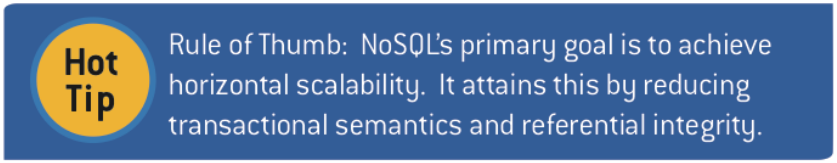
Examples: Neo4J, InfoGrid, Infinite Graph Typical applications: Social networking, Recommendations Strengths: Graph algorithms e.g. shortest path, connectedness, n degree relationships, etc. Weaknesses: Has to traverse the entire graph to achieve a definitive answer. Not easy to cluster.

NoSQL is a large and expanding field, for the purposes of this paper the common features of NoSQL data stores are:

- Easy to use in conventional load-balanced clusters
- Persistent data (not just caches)
- Scale to available memory
- Have no fixed schemas and allow schema migration without downtime
- Have individual query systems rather than using a standard query language
- Are ACID within a node of the cluster and eventually consistent across the cluster

Distributed Peer Stores

Examples: Cassandra, HBase, Riak Typical applications: Distributed file systems Strengths: Fast lookups, good distributed storage of data Weaknesses: Very low-level API .



Document databases

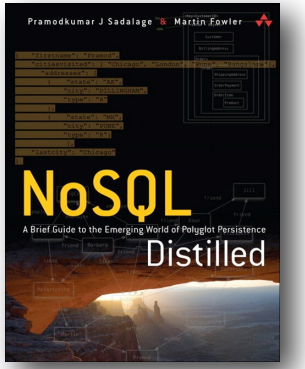
Examples: CouchDB, MongoDb Typical applications: Web applications Strengths: Tolerant of incomplete data Weaknesses: Query performance, no standard query syntax

Object stores

Examples: Oracle Coherence, db4o, ObjectStore, GemStone, Polar Typical applications: Finance systems Strengths: Matches OO development paradigm, low-latency ACID, mature technology Weaknesses: Limited querying or batch-update options

http://www.thoughtworks.com/insights/articles/nosql-comparison#Types-of-NoSQL-datastores

NoSQL Key Points



• Relational databases have been a successful technology for twenty years, providing persistence, concurrency control, and an integration mechanism.

• Application developers have been frustrated with the impedance mismatch between the relational model and the in-memory data structures.

• There is a movement away from using databases as integration points towards encapsulating databases within applications and integrating through services.

• The vital factor for a change in data storage was the need to support large volumes of data by running on clusters. Relational databases are not designed to run efficiently on clusters.

• NoSQL is an accidental neologism. There is no prescriptive definition—all you can make is an observation of common characteristics.

• The common characteristics of NoSQL databases are

  • Not using the relational model

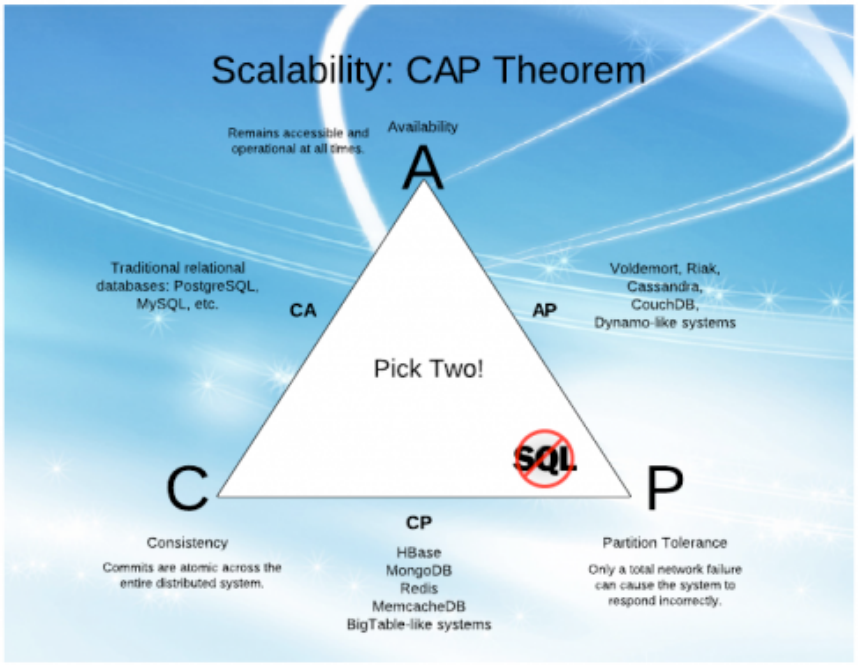
  • Running well on clusters

  • Open-source

  • Built for the 21st century web estates

  • Schemaless

  • The most important result of the rise of NoSQL is Polyglot Persistence.



The CAP theorem asserts that any networked shared-data system can have only two of three desirable properties.

However, by explicitly handling partitions, designers can optimize consistency and availability, thereby achieving some trade-off of all three.

Eric Brewer, *University of California, Berkeley*

Modern CAP

CAP prohibits only a tiny part of the design space: perfect availability and consistency in the presence of partitions, which are rare.

When partitions are present, there is an incredible range of flexibility for handling partitions and recovering from them.

The modern CAP goal should be to maximize combinations of consistency and avail- ability that make sense for the specific application.

1. Partition Mode Operation
2. Partition Recovery

2 out of 3

The easiest way to understand CAP is to think of two nodes on opposite sides of a partition.

Allowing at least one node to update state will cause the nodes to become inconsistent, thus forfeiting C.

Likewise, if the choice is to preserve consistency, one side of the partition must act as if it is unavailable, thus forfeiting A.

Only when nodes communicate is it possible to preserve both consistency and availability, thereby forfeiting P.

CAP, BASE, ACID

The general belief is that for wide-area systems, designers cannot forfeit P and therefore have a difficult choice between C and A.

The NoSQL movement is about creating choices that focus on availability first and consistency second.

Databases that adhere to ACID properties (atomicity, consistency, isolation, and durability) do the opposite.

CAP Confusion

"2 of 3" view is misleading on several fronts.

First, because **partitions are rare**, there is little reason to forfeit C or A when the system is not partitioned.

Second, the choice between C and A can occur many times within the same system at very fine granularity; not only can **subsystems** make different choices, but the choice can change according to the **operation** or even the specific **data** or user involved.

Finally, **all three properties are more continuous than binary**. Availability is obviously continuous from 0 to 100 percent, but there are also many levels of consistency, and even partitions have nuances, including disagreement within the system about whether a partition exists

Manage the Partitions

The challenging case for designers is to mitigate a partition's effects on consistency and availability. The key idea is to manage partitions very explicitly, including not only detection, but also a specific recovery process and a plan for all of the invariants that might be violated during a partition.

Three Steps:

1. Detect the start of a partition
2. Enter an explicit partition mode that may limit some operations
3. Initiate partition recovery when communication is restored

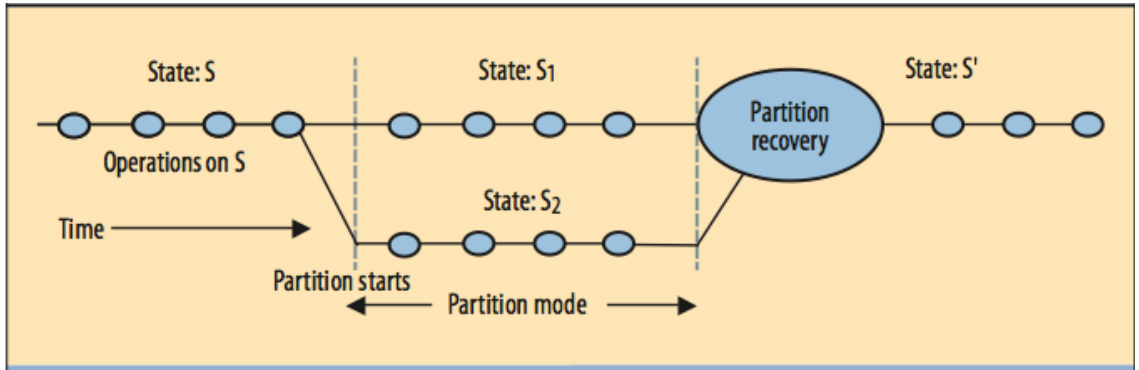


Figure 1. The state starts out consistent and remains so until a partition starts. To stay available, both sides enter partition mode and continue to execute operations, creating concurrent states S_1 and S_2, which are inconsistent. When the partition ends, the truth becomes clear and partition recovery starts. During recovery, the system merges S_1 and S_2 into a consistent state S' and also compensates for any mistakes made during the partition.

Partition Recovery

The designer must solve two hard problems during recovery:

- The state on both sides must become consistent, and
- There must be compensation for the mistakes made during partition mode.

*Using **commutative operations** is the closest approach to a general framework for automatic state convergence.*

The system concatenates logs, sorts them into some order, and then executes them. Commutativity implies the ability to rearrange operations into a preferred consistent global order. Unfortunately, using only commutative operations is harder than it appears; for example, addition is com- mutative, but addition with a bounds check is not (a zero balance, for example)

The ATM system designer could choose to prohibit withdrawals during a partition, since it is impossible to know the true balance at that time, but that would compromise availability. Instead, using stand-in mode (partition mode), modern ATMs limit the net withdrawal to at most k, where k might be $200. Below this limit, withdrawals work completely; when the balance reaches the limit, the system denies withdrawals. Thus, the ATM chooses a sophisticated limit on availability that permits withdrawals but bounds the risk.

*When the partition ends, there must be some way to both restore consistency and compensate for mistakes made while the system was partitioned. Restoring state is easy because the operations are commutative, but **compensation** can take several forms. A final balance below zero violates the invariant. In the normal case, the ATM dispensed the money, which caused the mistake to become external. The bank compensates by charging a fee and expecting repayment.*

ThoughtWorks®

Technology Radar

Prepared by the ThoughtWorks Technology Advisory Board - May 2013

"For problems that fit the document database model, ***MongoDB*** *is now the most popular choice"*

Platforms

Adopt

29. Elastic Search
30. MongoDB
31. Neo4J
32. Redis
33. SMS and USSD as a UI

Trial

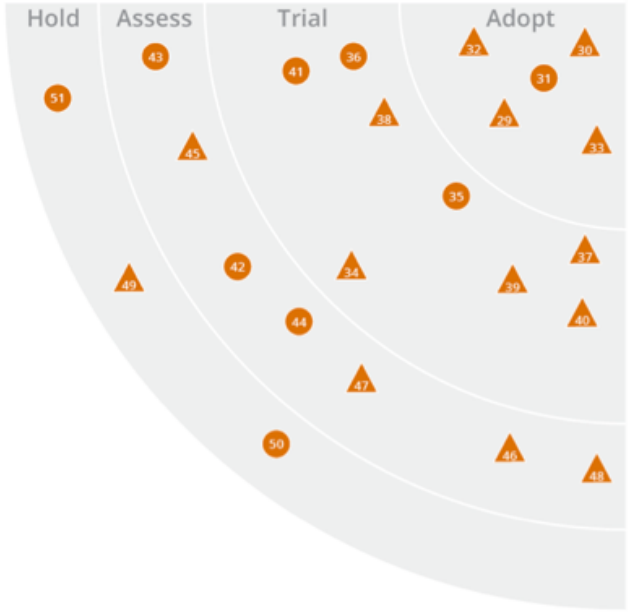
34. BigQuery
35. Continuous integration in the cloud
36. Couchbase
37. Hadoop 2.0
38. Node.js
39. OpenStack
40. Rackspace Cloud
41. Riak

Assess

42. Azure
43. Calatrava
44. Datomic
45. PhoneGap/Apache Cordova
46. PostgreSQL for NoSQL
47. Vumi
48. Zepto.js

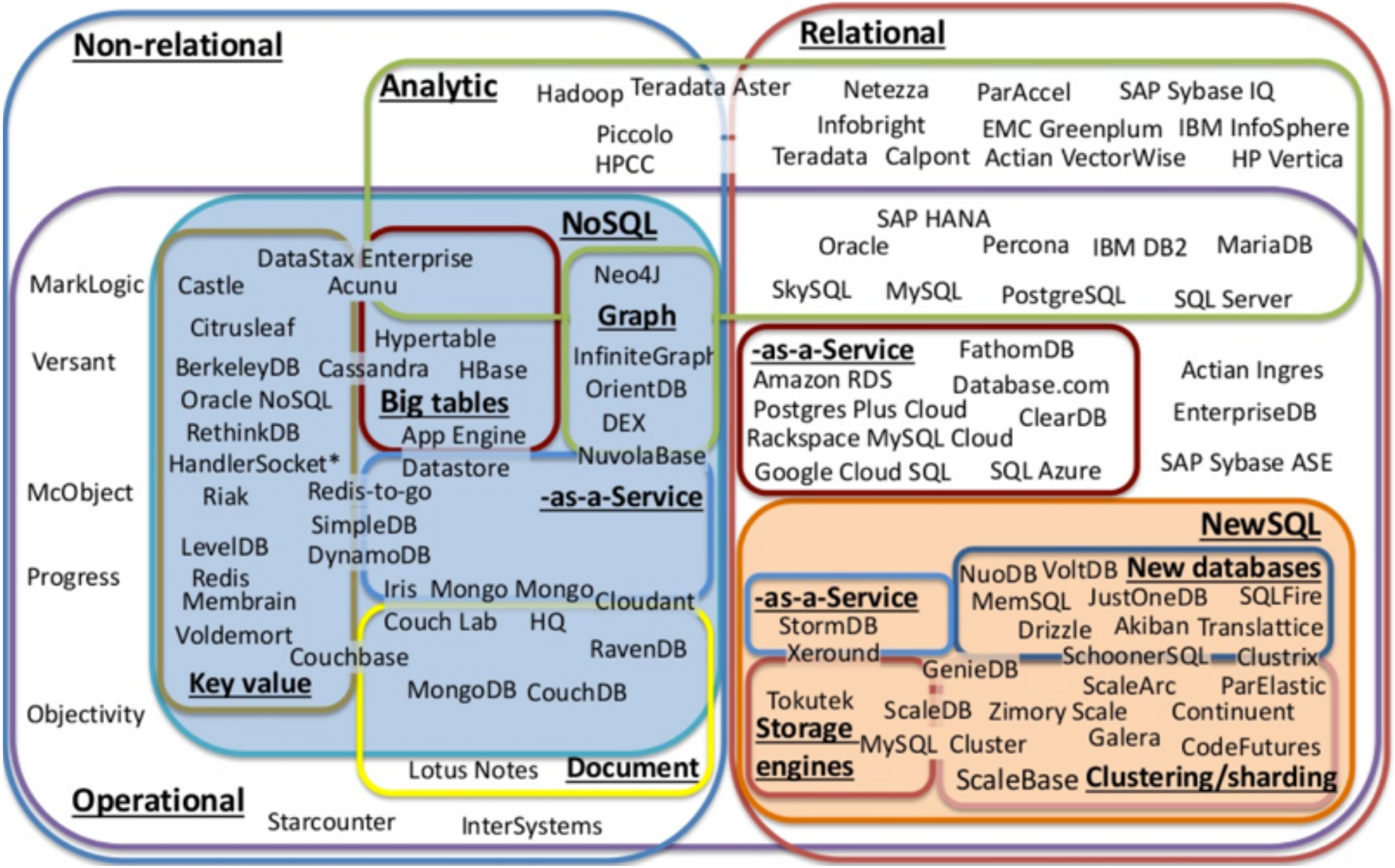
Hold

49. Big enterprise solutions
50. Singleton infrastructure
51. WS-*



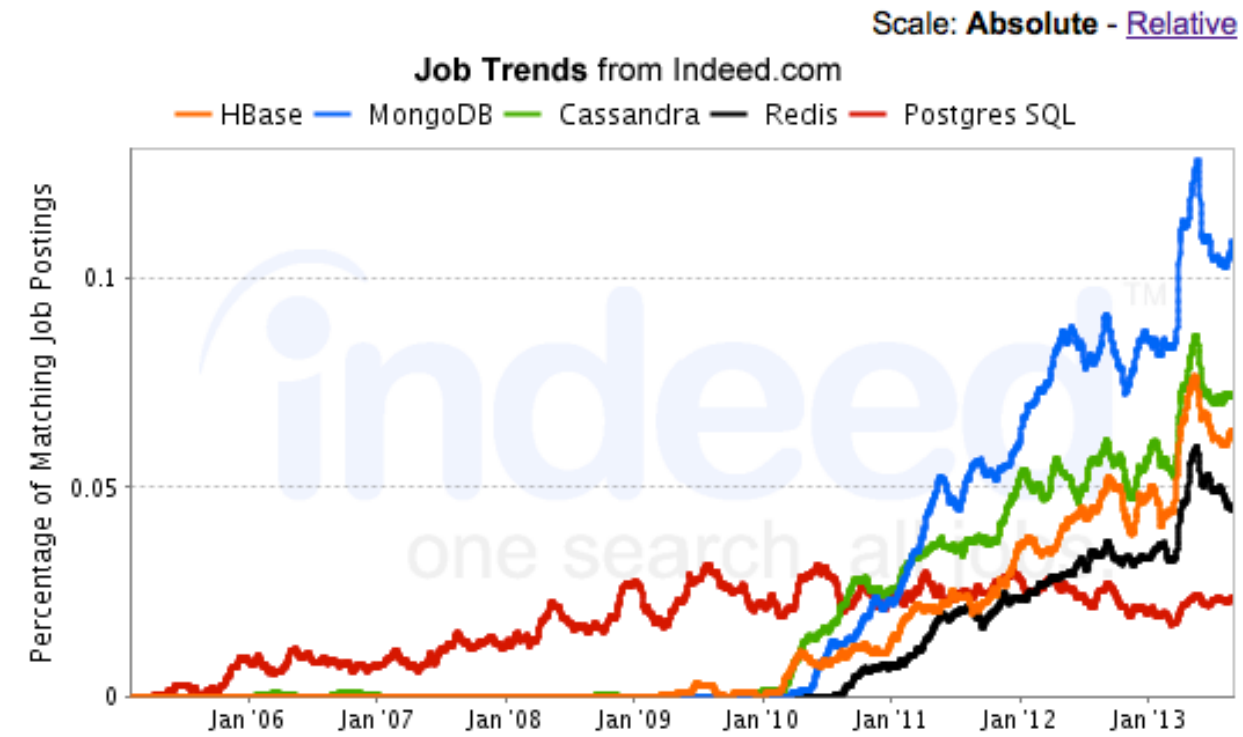
http://www.thoughtworks.com/radar

*"**PostgreSQL** is expanding to become the NoSQL choice of SQL databases"*



http://www.slideshare.net/ScaleBase/choosing-a-next-gen-database-the-new-world-order-of-no-sql-newsql-and-mysql

HBase, MongoDB, Cassandra, Redis, Postgres SQL Job Trends



Indeed.com searches millions of jobs from thousands of job sites.
This job trends graph shows the percentage of jobs we find that contain your search terms.

Find Hbase jobs, Mongodb jobs, Cassandra jobs, Redis jobs, Postgres SQL jobs

Feel free to share this graph

Email to a friend

Post on your blog/website

Top Job Trends

1. HTML5
2. MongoDB
3. iOS
4. Android
5. Mobile app
6. Puppet
7. Hadoop
8. jQuery
9. PaaS
10. Social Media

http://www.indeed.com/jobtrends

Aggregate Data Models

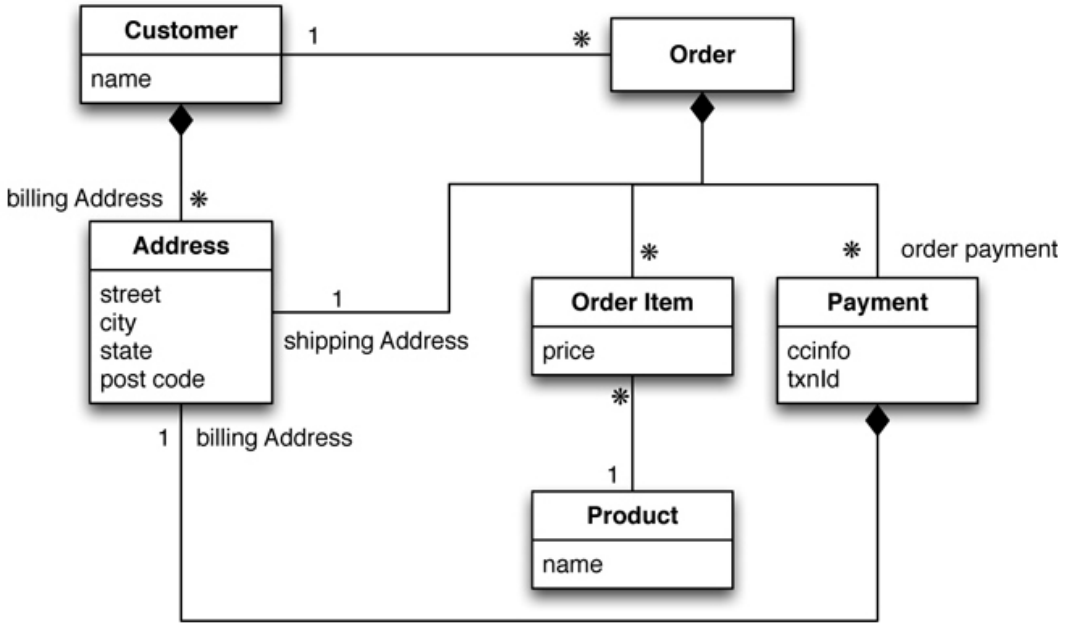
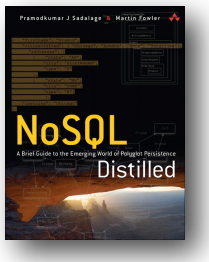


Figure 2.3. An aggregate data model

```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
  "productId":27,
  "price": 32.45,
  "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```



"NoSQL Databases support atomic manipulation of a single aggregate at a time. This means that if we need to manipulate multiple aggregates in an atomic way, we have to manage that ourselves in the application code"

NoSQL Data Models

Example of a standard structured (relational) database table for user records

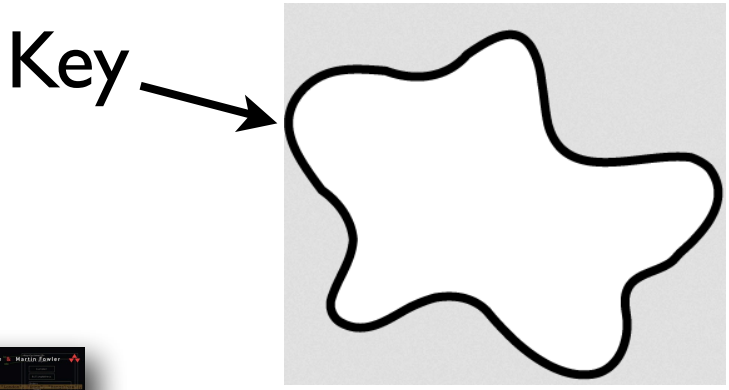
ID	First Name	Last Name
1	Samantha	Jones
2	James	Bond
3	James	Kirk

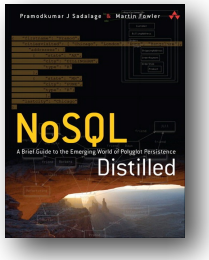
Example of unstructured data for user records

Key: 1	ID: sj	First Name: Sam

Key: 2	Email: jb@gmail.com	Location: London	Age: 37

Key: 3	Facebook ID: jkirk	Password: xxx	Name: James






```
// in customers
{
"id":1,
"name":"Martin",
"billingAddress":[{"city":"Chicago"}]
}

// in orders
{
"id":99,
"customerId":1,
"orderItems":[
  {
   "productId":27,
   "price": 32.45,
   "productName": "NoSQL Distilled"
    }
  ],
"shippingAddress":[{"city":"Chicago"}]
"orderPayment":[
  {
    "ccinfo":"1000-1000-1000-1000",
    "txnId":"abelif879rft",
    "billingAddress": {"city": "Chicago"}
  }
  ],
}
```

NoSQL Data Models

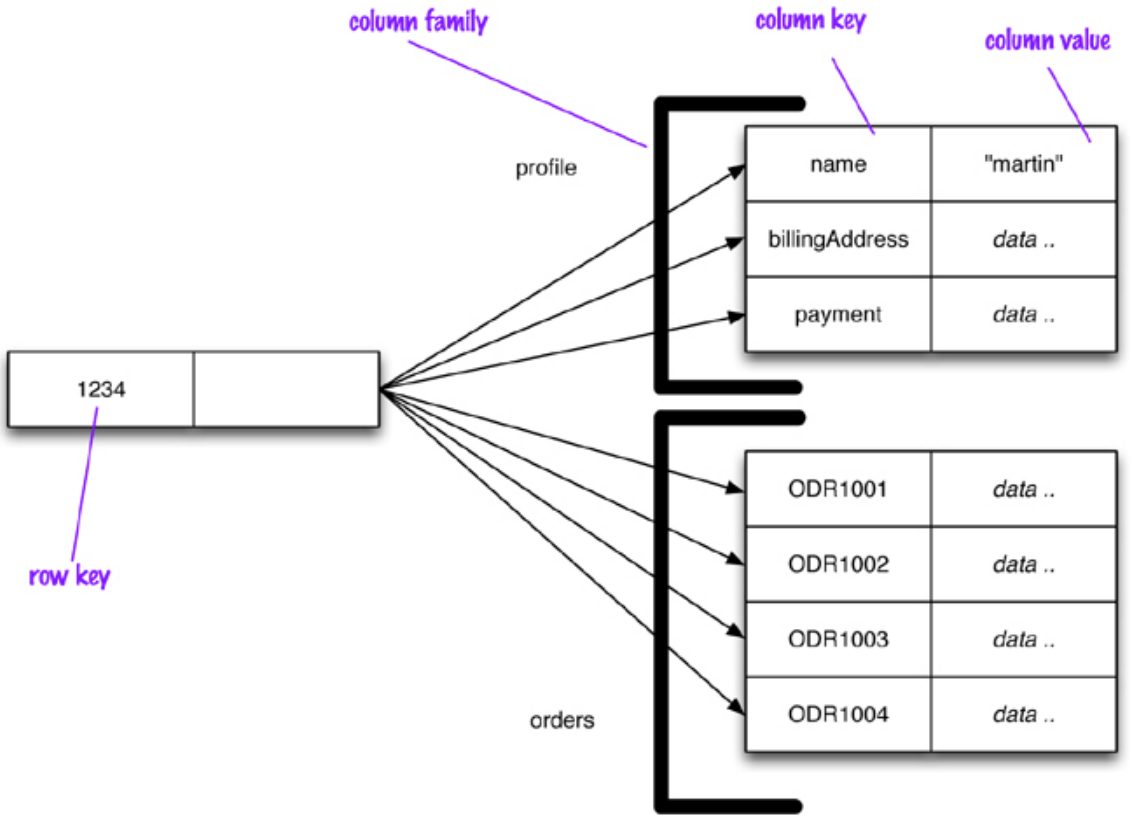
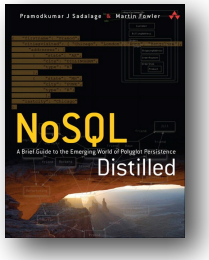
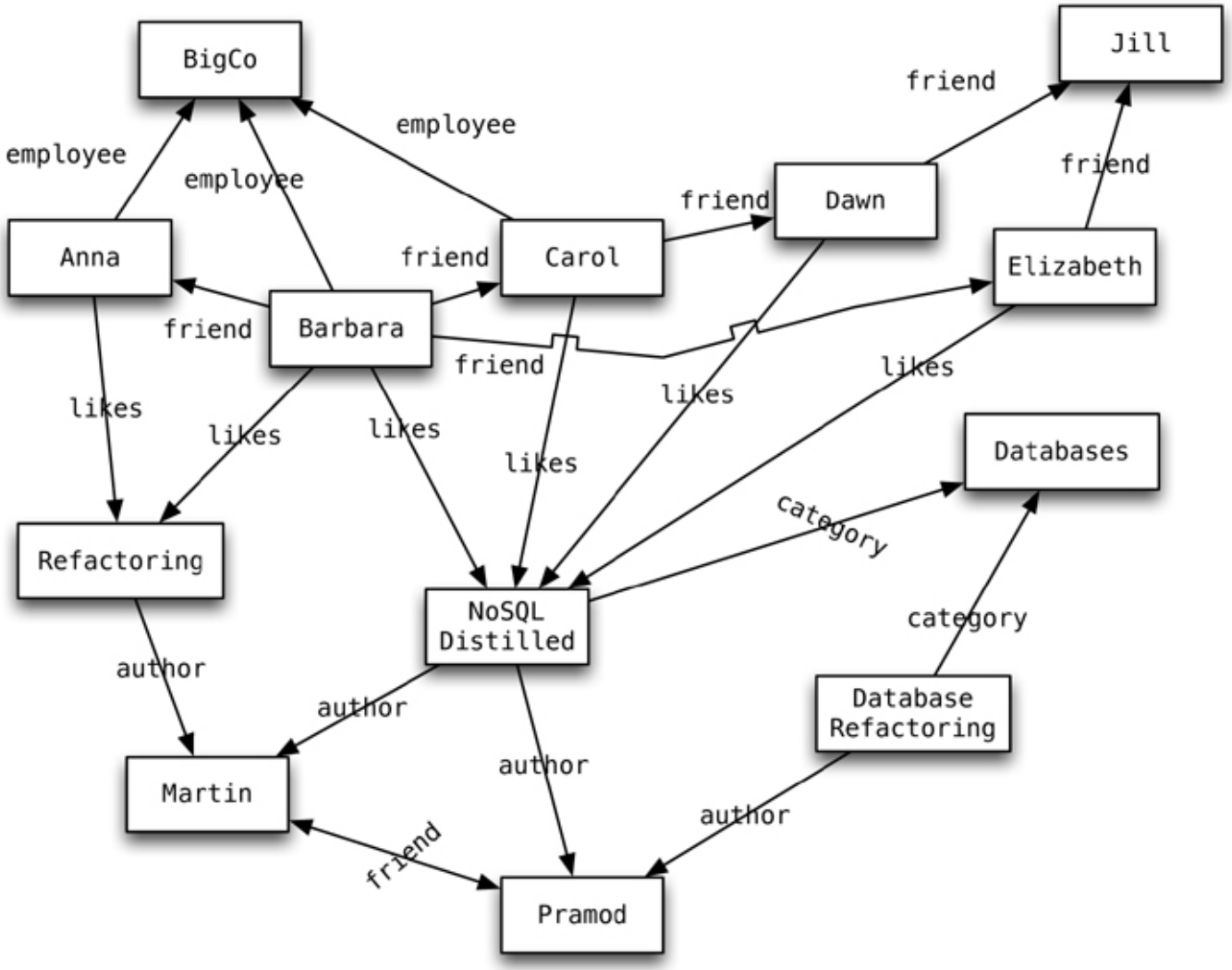


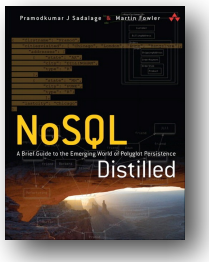
Figure 2.5. Representing customer information in a column-family structure



"Column-family databases organize their columns into column families. Each column has to be part of a single column family, and the column acts as unit for access, with the assumption that data for a particular column family will be usually accessed together."

Graph Databases



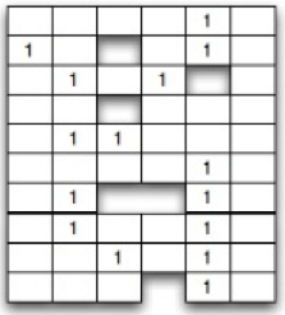


NoSQL Data Models

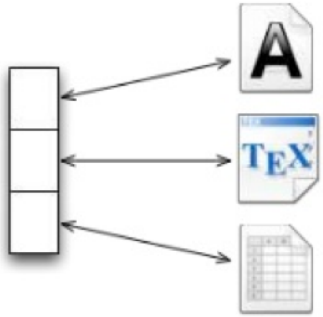
Key-Value

Graph DB

BigTable



Document



Distributions Models

- Single-Server

-- Using NoSQL for Data Model

- Master-Slave Replication

-- Same Data is replicated from Master to Slaves. Master services all Writes. Reads services from Masters or Slaves.

- Sharding

-- Different Data is on separate nodes, each of which does its own Reads and Writes.

- Peer-To-Peer Replication

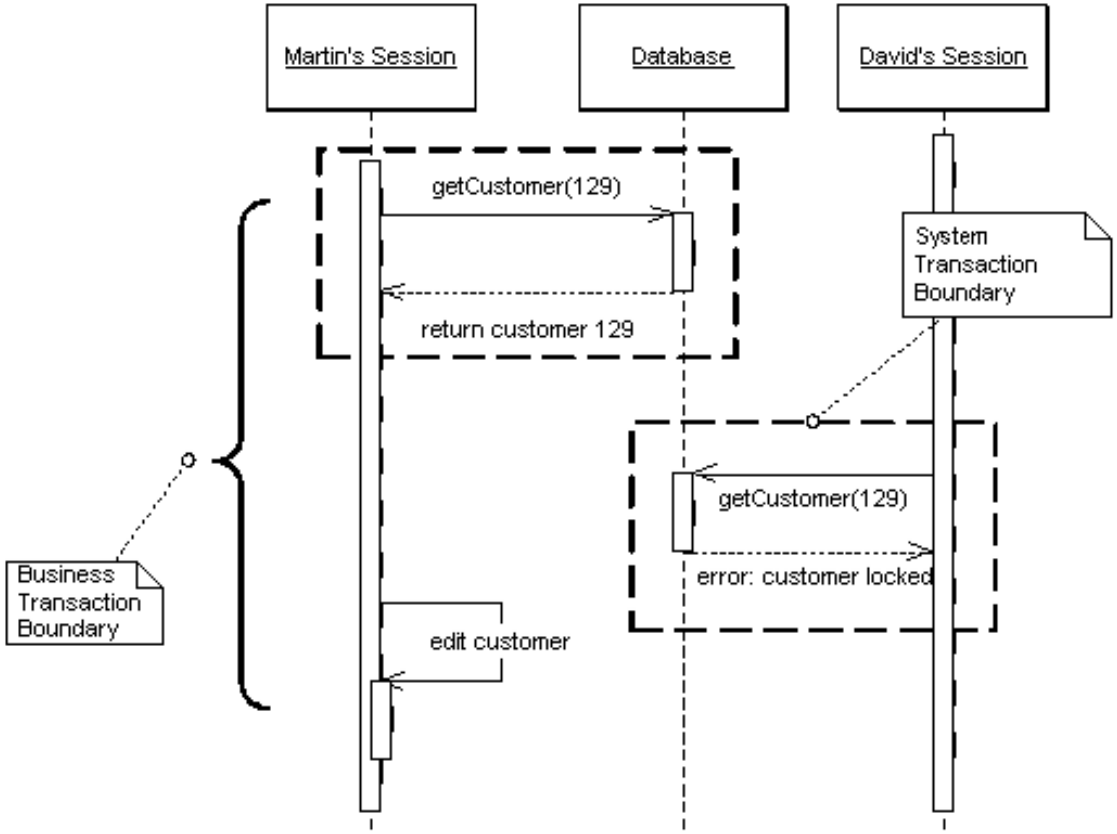
-- All Nodes services Reads and Writes to All Data.

Pessimistic Offline Lock

by David Rice

Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.

For a full description see P of EAA page **426**



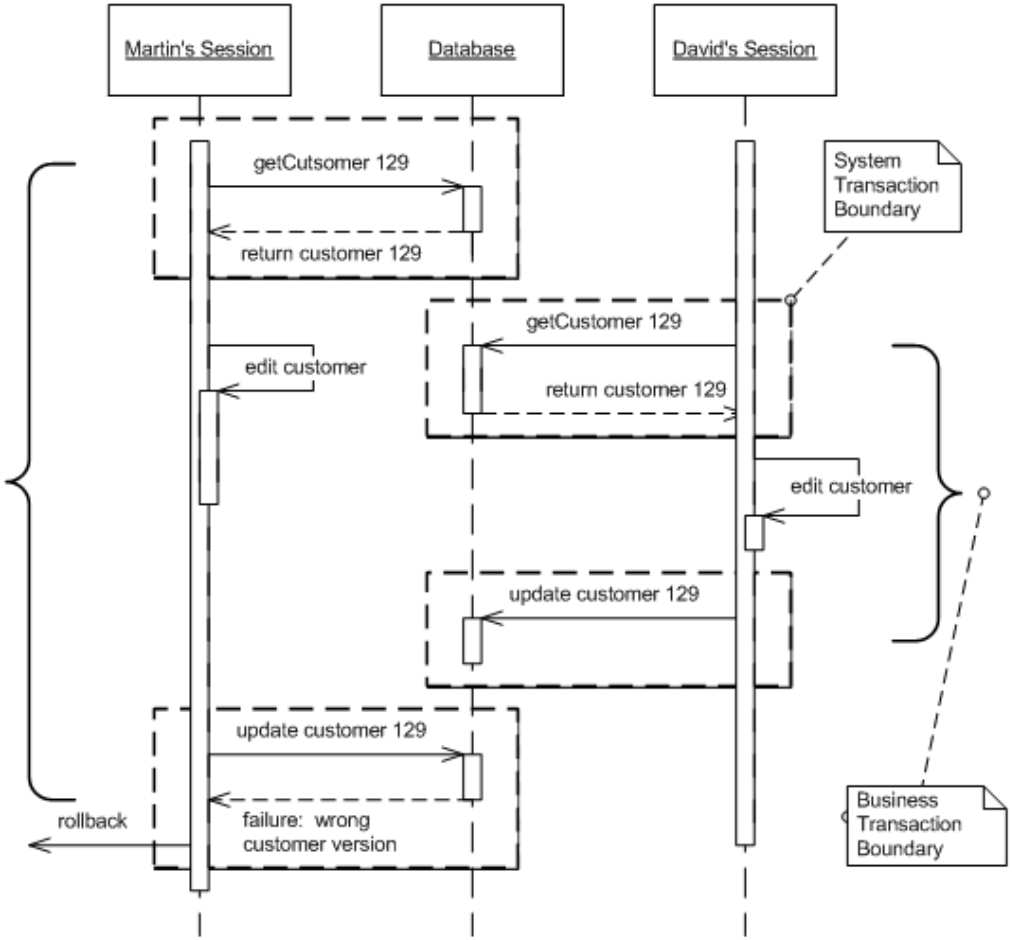
http://martinfowler.com/eaaCatalog/pessimisticOfflineLock.html

Optimistic Offline Lock

by David Rice

Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.

For a full description see P of EAA page **416**



http://martinfowler.com/eaaCatalog/optimisticOfflineLock.html

Version Stamps

A field that changes each time data is updated

Examples:

Counter	Requires a Single Master to generate unique values.
GUID	Anyone can generate. But values are large and hard to compare.
Hash	Hash of content with a big enough hash key size for global uniqueness.
Timestamp	Nodes have to synchronize clocks.
Vector Stamps	*A vector of version stamps from each node.* *Example: [Node-A: 2, Node-B: 5, Node-C: 10]*

Dynamo

Optimized for Availability

Key-Value (Opaque)

Consistent Hashing (Ring System)

Peer-To-Peer Replication

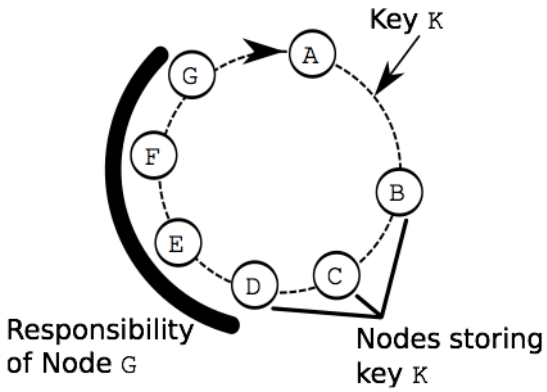


Figure 2.2: Key K is stored with a replication count of 3. It is stored on node B, while copies are kept on nodes C and D. Node G is responsible for all keys from D to G (left black arc).

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Big Table

Optimized for Consistency

Key/Record (Column-Oriented)

Master-Slave Replication

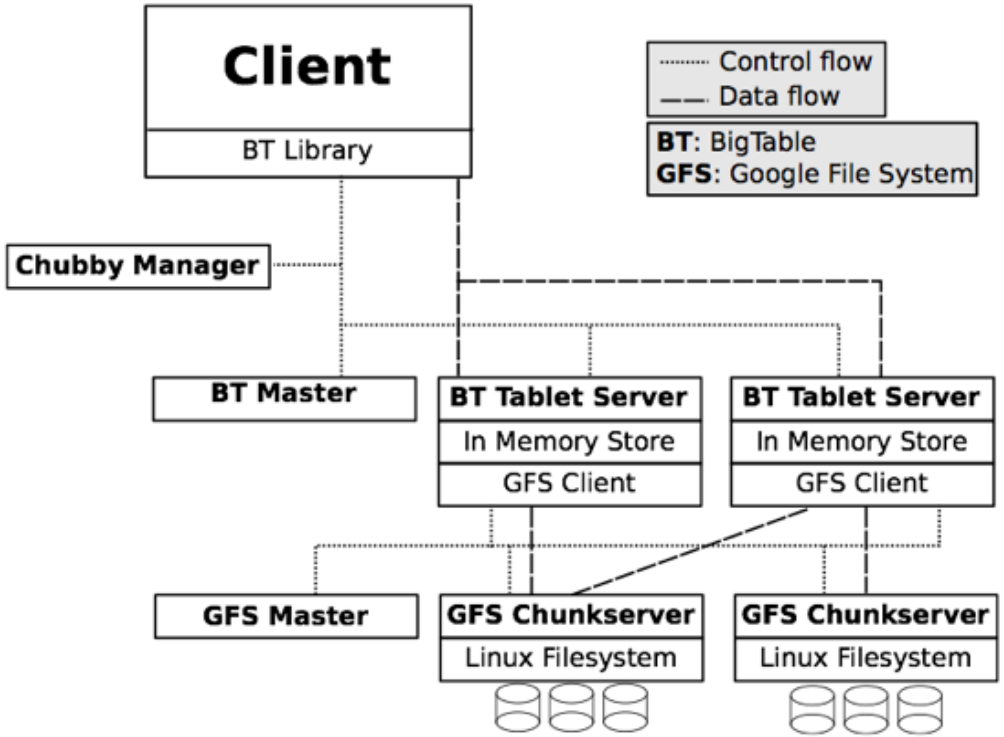


Figure 2.3: Architecture of Google BigTable: BigTable (BT) is created on top of two existing components: Google File System (GFS) and the Chubby lock service. While the control flow (dotted lines) is mainly between clients and the managers, the data (dashed lines) is transferred between the clients and the providers without the master.

Comparison of Distribution Technologies in Different NoSQL Database Systems

	Programming Language	License	First Release
Apache Cassandra	Java	Apache License 2	July 2008
Basho Riak	Erlang, JavaScript	Apache License 2	2009
Project Voldermort	Java	Apache License 2	Feburary 2009
Apache HBase	Java	Apache License 2	2007
Redis	C	New BSD	March 2009
CouchDB	Erlang, JavaScript	Apache License 2	2005
MongoDB	C++	GNU AGPL v3.0	Summer 2008
Membase	C++, Erlang	Apache License 2	2009

Table 4.1: Overview of the system which were examined

	Replication	Fragmentation	Category
Cassandra	✓	✓	Ring (4.2)
Riak	✓	✓	
Project Voldemort	✓	✓	
HBase	*by underlying DFS*	✓	Master-Slave (4.3)
MongoDB	✓	✓	
Membase	✓	✓	
Redis	✓*(unidirectional)*	*using external tools*	Replication based (4.4)
CouchDB	✓*(bidirectional)*	*using external tools*	

Table 4.2: Categorization and features of the systems

	Column-Oriented	Document-Oriented	Schemaless
Ring	Cassandra		Riak, Project Voldemort
Master-Slave	HBase	MongoDB	Membase
Replication based		CouchDB	Redis

Table 5.1: Systems compared in regard to data models and distribution categories

	Ring	Master-Slave	Replication Based
Single Point Of Failure	none	master and slaves	unidirectional: the master otherwise: none
Consistency Model	tunable consistency	strict consistency	eventual consistency
Availability	write: always available read: maybe unavailable	maybe unavailable	unidirectional: Maybe unavailble bidirectional: always available
Data Access	range scans make no sense (if randomly placed)	range scans fast (same slave)	range scans fast (all data available)

Table 4.3: Differences of the categories which have been discussed above

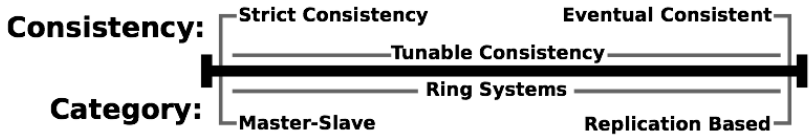


Figure 4.1: The categories introduced directly correlate with the consistency models they implement if put on a scale between strict consistency on the one side and eventual consistency on the other.

Map Reduce

http://nosqltapes.com/video/understanding-mapreduce-with-mike-miller

Polymorphic Application

- **Current Approach**

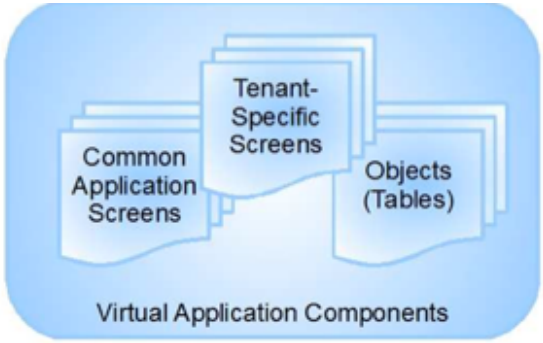
  - MetaData Driven

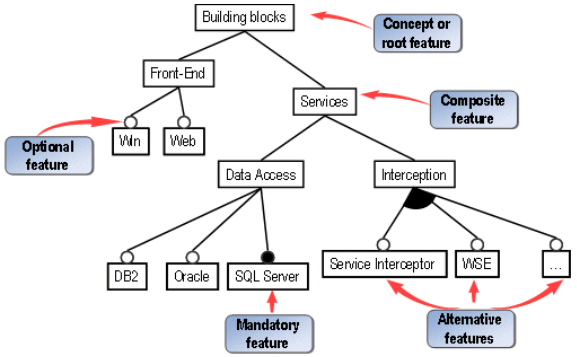
  1. Dedicated Database
  2. Shared DB, Fixed Extension Set
  3. Shared DB, Custom Extensions

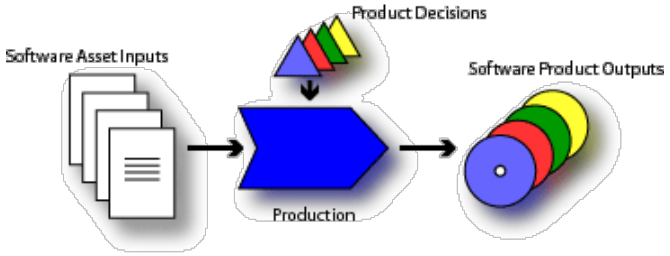
- **Emerging Approach:**

  - ByteCode Injection

  1. Beyond Objects
  2. Aspect-Oriented Weaving
  3. Product-Line Engineering







Polyglot Programming

- **Current Approach**

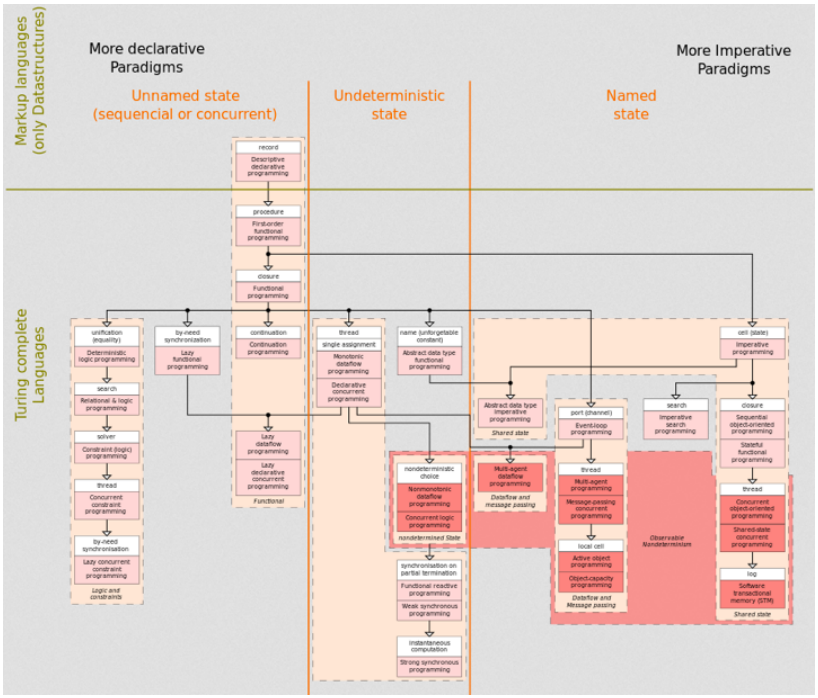
  - *Single Paradigm Design*

  1. Object-Oriented Analysis
  2. Object-Oriented Design
  3. Object-Oriented Programming

- **Emerging Approach:**

  - *Multi-Paradigm Design*

  1. Beyond Objects
  2. Multiple Languages (GSL + DSL)
  3. OOP + Functional + AOP



http://polyglotprogramming.com/

http://en.wikipedia.org/wiki/File:Programming_paradigms.svg

Polyglot Persistence

- **Current Approach**

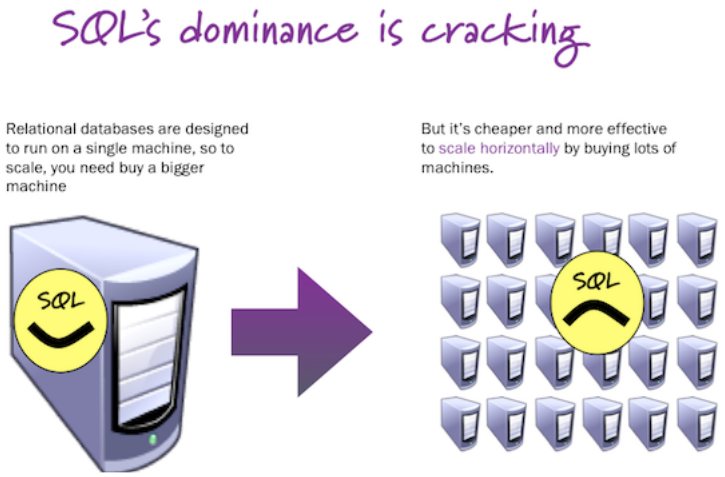
  - Relational is King!

  1. OLTP (Transactions)
  2. OLAP (Analytics)
  3. Centralized DB Server
  4. Scale-Up (add expensive hardware)

- **Emerging Approach:**

  - Not Just Relational

  1. SQL + NoSQL (Eventual Consistency)
  2. Hadoop + Columnar DB (Analytics)
  3. Distributed Data Stores
  4. Scale-Out (add cheap computers)

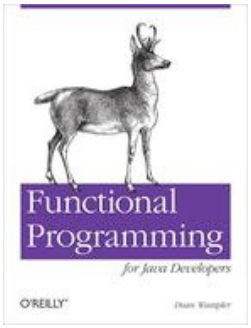


http://martinfowler.com/bliki/PolyglotPersistence.html

Map & Reduce in Functional Programming

Two important concepts in functional programming:

1. **Map**: do something to everything in a list
2. **Fold**: combine results of a list in some way



Combinator Functions: The Collection Power Tools

You already think of lists, maps, etc. as "collections," all with a set of common methods. Most collections support Java `Iterators`, too. In functional programming, there are three core operations that are the basis of almost all work you do with collections:

Filter

> Create a new collection, keeping only the elements for which a filter method returns `true`. The size of the new collection will be less than or equal to the size of the original collection.

Map

> Create a new collection where each element from the original collection is transformed into a new value. Both the original collection and the new collection will have the same size. (Not to be confused with the `Map` data structure.)

Fold

> Starting with a "seed" value, traverse through the collection and use each element to build up a new final value where each element from the original collection "contributes" to the final value. An example is summing a list of integers.

Many other common operations can be built on top of these three. Together they are the basis for implementing concise and *composable* behaviors. Let's see how.

http://www.joelonsoftware.com/items/2006/08/01.html

Joel on Software

Can Your Programming Language Do This?

by Joel Spolsky

Tuesday, August 01, 2006

One day, you're browsing through your code, and you notice two big blocks that look almost exactly the same. In fact, they're exactly the same, except that one block refers to "Spaghetti" and one block refers to "Chocolate Moose."

```
// A trivial example:

alert("I'd like some Spaghetti!");
alert("I'd like some Chocolate Moose!");
```

These examples happen to be in JavaScript, but even if you don't know JavaScript, you should be able to follow along.

The repeated code looks wrong, of course, so you create a function:

SQL vs. Map Reduce

Below is a selection from the "Orders" table:

OrderID	CustomerID	EmployeeID	OrderDate	ShipperID
10248	90	5	1996-07-04	3
10249	81	6	1996-07-05	1
10250	34	4	1996-07-08	2

And a selection from the "Shippers" table:

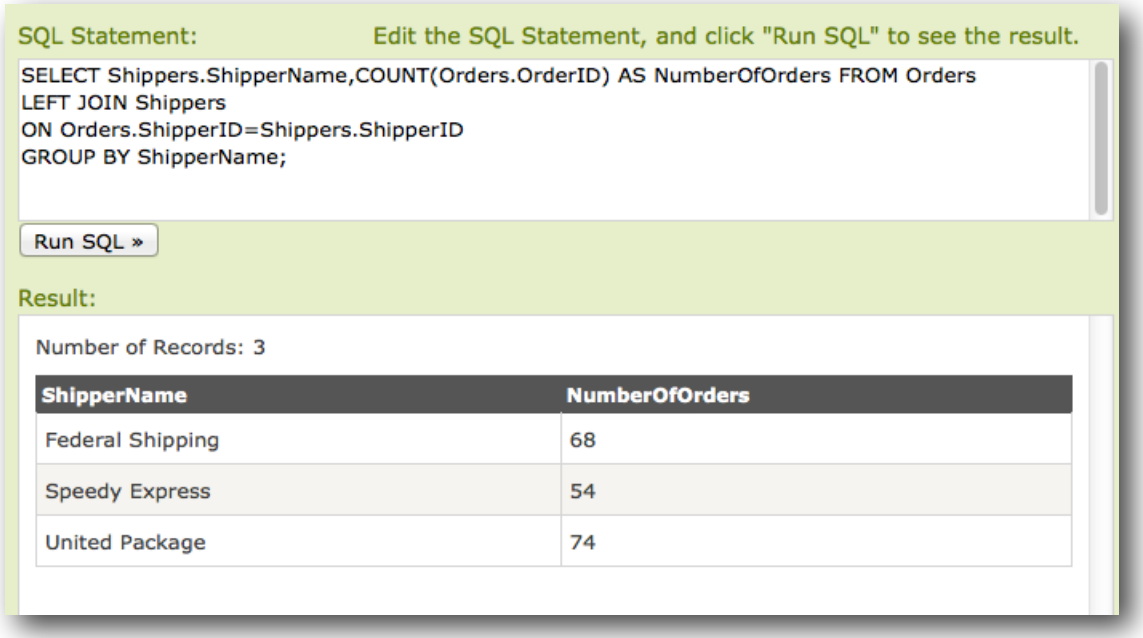
ShipperID	ShipperName	Phone
1	Speedy Express	(503) 555-9831
2	United Package	(503) 555-3199
3	Federal Shipping	(503) 555-9931

And a selection from the "Employees" table:

EmployeeID	LastName	FirstName	BirthDate	Photo	Notes
1	Davolio	Nancy	1968-12-08	EmpID1.pic	Education includes a BA....
2	Fuller	Andrew	1952-02-19	EmpID2.pic	Andrew received his BTS....
3	Leverling	Janet	1963-08-30	EmpID3.pic	Janet has a BS degree....

http://www.w3schools.com/sql/sql_groupby.asp

SQL vs. Map Reduce



SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName,COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
GROUP BY ShipperName;
```

Run SQL »

Result:

Number of Records: 3

ShipperName	NumberOfOrders
Federal Shipping	68
Speedy Express	54
United Package	74

How does a SQL Query Engine Process this Query?

How would this be done in Map Reduce?

http://www.w3schools.com/sql/sql_groupby.asp

Breaking Down Query Processing

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName, Orders.OrderID, Orders.CustomerID FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
```

Run SQL »

Result:

Number of Records: 196

ShipperName	OrderID	CustomerID
Federal Shipping	10248	90
Speedy Express	10249	81
United Package	10250	34
Speedy Express	10251	84
United Package	10252	76
United Package	10253	34
United Package	10254	14
Federal Shipping	10255	68
United Package	10256	88
Federal Shipping	10257	35
Speedy Express	10258	20
Federal Shipping	10259	13
Speedy Express	10260	55
United Package	10261	61

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName, 1 FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
```

Run SQL »

Result:

Number of Records: 196

ShipperName	Expr1001
Federal Shipping	1
Speedy Express	1
United Package	1
Speedy Express	1
United Package	1
United Package	1
United Package	1
Federal Shipping	1
United Package	1
Federal Shipping	1
Speedy Express	1
Federal Shipping	1
Speedy Express	1
United Package	1

http://www.w3schools.com/sql/sql_groupby.asp

Breaking Down Query Processing

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName, SUM(1) FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
GROUP BY ShipperName
```

Run SQL »

Result:

Number of Records: 3

ShipperName	Expr1001
Federal Shipping	68
Speedy Express	54
United Package	74

These are Equivalent!

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName,COUNT(Orders.OrderID) AS NumberOfOrders FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
GROUP BY ShipperName;
```

Run SQL »

Result:

Number of Records: 3

ShipperName	NumberOfOrders
Federal Shipping	68
Speedy Express	54
United Package	74

http://www.w3schools.com/sql/sql_groupby.asp

What's Going on Here?

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName, 1 FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
```

Relational Projections on Base Tables

Number of Records: 196

ShipperName	Expr1001
Federal Shipping	1
Speedy Express	1
United Package	1
Speedy Express	1
United Package	1
United Package	1
United Package	1
Federal Shipping	1
United Package	1
Federal Shipping	1
Speedy Express	1
Federal Shipping	1
Speedy Express	1
United Package	1

In Memory Temp Table

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName, SUM(1) FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
GROUP BY ShipperName
```

Relational Aggregations

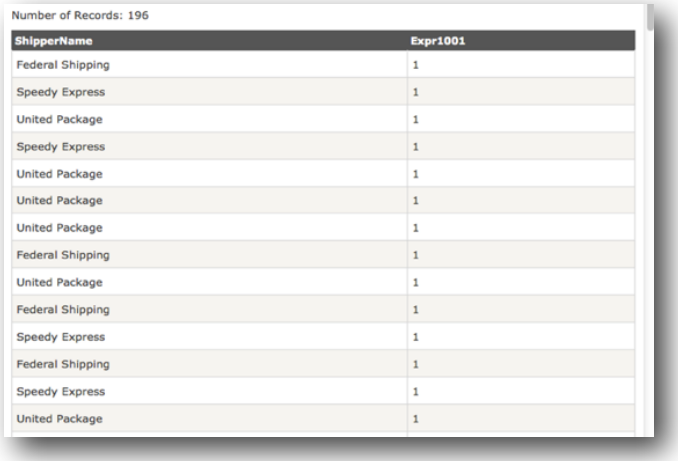
ShipperName	Expr1001
Federal Shipping	68
Speedy Express	54
United Package	74

Result (Output)

The Map Reduce Way

```
OrderID,ShipperName,CustomerID,OrderDate
10248, Federal Shipping, 90, 7/4/1996
10249, Speedy Express, 81, 7/5/1996
10250, United Package, 34, 7/8/1996
```

Map Input
(Key = OrderID,
Values = Line)



Number of Records: 196

ShipperName	Expr1001
Federal Shipping	1
Speedy Express	1
United Package	1
Speedy Express	1
United Package	1
United Package	1
United Package	1
Federal Shipping	1
United Package	1
Federal Shipping	1
Speedy Express	1
Federal Shipping	1
Speedy Express	1
United Package	1

Intermediate Results
(Key = Shipper, Value = 1)

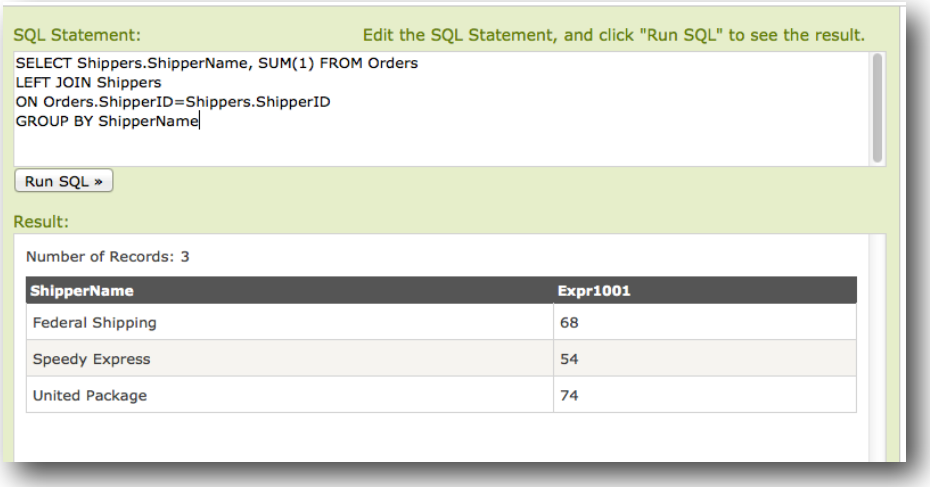
***Reduce** Intermediate Results*
Add all the 1's for each Shipper
(Key = Shipper, Value = Sum of Ones')

ShipperName	Expr1001
Federal Shipping	68
Speedy Express	54
United Package	74

Result (Output)

Which is Easier for Developers?

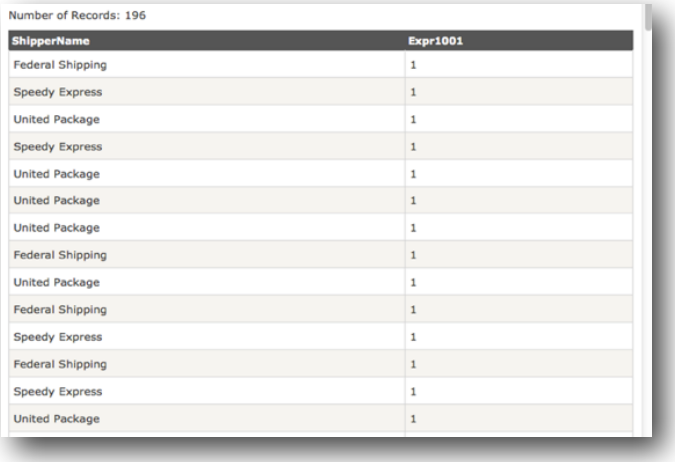
Write SQL Query



Get Results!

```
OrderID,ShipperName,CustomerID,OrderDate
10248, Federal Shipping, 90, 7/4/1996
10249, Speedy Express, 81, 7/5/1996
10250, United Package, 34, 7/8/1996
```

Write Map Function (Java?)



Write Reduce Function (Java?)

ShipperName	Expr1001
Federal Shipping	68
Speedy Express	54
United Package	74

Which can Scale?

Terabytes of Data across Thousands of Servers

```
OrderID,ShipperName,CustomerID,OrderDate
10248, Federal Shipping, 90, 7/4/1996
10249, Speedy Express, 81, 7/5/1996
10250, United Package, 34, 7/8/1996
```

Map

1 Million Rows

SQL

SQL Statement: Edit the SQL Statement, and click "Run SQL" to see the result.

```
SELECT Shippers.ShipperName, SUM(1) FROM Orders
LEFT JOIN Shippers
ON Orders.ShipperID=Shippers.ShipperID
GROUP BY ShipperName
```

Run SQL »

Result:

Number of Records: 3

ShipperName	Expr1001
Federal Shipping	68
Speedy Express	54
United Package	74

Number of Records: 196

ShipperName	Expr1001
Federal Shipping	1
Speedy Express	1
United Package	1
Speedy Express	1
United Package	1
United Package	1
United Package	1
Federal Shipping	1
United Package	1
Federal Shipping	1
Speedy Express	1
Federal Shipping	1
Speedy Express	1
United Package	1

Reduce

ShipperName	Expr1001
Federal Shipping	68
Speedy Express	54
United Package	74

Examples from Paper

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Inverted Index: The map function parses each document, and emits a sequence of ⟨word, document ID⟩ pairs. The reduce function accepts all pairs for a given word, sorts the corresponding document IDs and emits a ⟨word, *list*(document ID)⟩ pair. The set of all output pairs forms a simple inverted index. It is easy to augment this computation to keep track of word positions.

Distributed Sort: The map function extracts the key from each record, and emits a ⟨key, record⟩ pair. The reduce function emits all pairs unchanged. This computation depends on the partitioning facilities described in Section 4.1 and the ordering properties described in Section 4.2.

Distributed Grep: The map function emits a line if it matches a supplied pattern. The reduce function is an identity function that just copies the supplied intermediate data to the output.

Count of URL Access Frequency: The map function processes logs of web page requests and outputs ⟨URL, 1⟩. The reduce function adds together all values for the same URL and emits a ⟨URL, total count⟩ pair.

Reverse Web-Link Graph: The map function outputs ⟨target, source⟩ pairs for each link to a target URL found in a page named source. The reduce function concatenates the list of all source URLs associated with a given target URL and emits the pair: ⟨target, *list*(source)⟩

Term-Vector per Host: A term vector summarizes the most important words that occur in a document or a set of documents as a list of ⟨*word*, *frequency*⟩ pairs. The map function emits a ⟨hostname, term vector⟩ pair for each input document (where the hostname is extracted from the URL of the document). The reduce function is passed all per-document term vectors for a given host. It adds these term vectors together, throwing away infrequent terms, and then emits a final ⟨hostname, term vector⟩ pair.

Map Reduce Execution Model