Database Revolution:
Old SQL, NewSQL, NoSQL …
Huh?

Prepared by Michael Bowers
2012-11-29
v. 2.8
Abstract

We are in the middle of a database revolution. NoSQL is disrupting the database world by innovating in three ways:

• New database architectures (software and hardware) that handle the large and ever growing velocity and volume of data dispersed across geographically distant data centers

• New graph and document modeling paradigms that compete with object, relational, and dimensional

• Schema-less databases to enable extreme agility of software development and rapid changes to huge data sets
What will you learn?

We are in the middle of a database revolution. NoSQL is disrupting the database world, but which NoSQL solution is best?

• Which one makes development fastest and easiest?
• Which one loads data the fastest?
• Which one scales the best?
• Which one queries data the best?
• Which one manages your data the best?
• Which one is the most mature?
• Which one is the least expensive?
• Which one should you use?

Facts -- no spin -- no agenda,
and lots of thought-provoking information.
About the Author

Michael Bowers

• Principal Architect
  LDS Church

• Author
  – *Pro CSS and HTML Design Patterns*
    • Published by Apress, 2007
  – *Pro HTML5 and CSS3 Design Patterns*
    • Published by Apress, 2011

• cssDesignPatterns.com
Agenda

• What is NoSQL?
• Which NoSQL is best?
Agenda

• What is NoSQL?
  – New Modeling Paradigms
  – New Hardware Paradigms

• Which NoSQL is best?
Agenda

• What is NoSQL?
  – New Modeling Paradigms
  – New Hardware Paradigms
• Which NoSQL is best?
What’s wrong with SQL DB’s?

• **Variability**
  – Relational modeling is not good at adapting to rapid change, which makes it incompatible with agile development

• **Variety**
  – Relational modeling organizes all data into tables, which is a suboptimal representation for graphs, hierarchies, cubes, linked-lists, unstructured data, etc.

• **Relevance**
  – Relational modeling isn’t good at relevance, which requires text and data to be stored in document context with links to and from other documents
Variability

Managing Rapid Change

• Schemas are incompatible with rapid change
  – Constantly evolving data structures
    • Can we afford to keep a large application in sync with regular changes to data structures?
  – Big data
    • Is data so large that it takes too long to modify values, structures, and indexes?
  – Agile development
    • Are requirements stable enough to create long-lasting relational data structures?

• Schemaless data is ideal for rapid change
  – Schemaless data and languages
    • XML, JSON, XQuery, SPARQL, JavaScript
  – Defensive programming is required
    • You never know what data queries will return
Variety

Handling data in all imaginable forms

- Impedance mismatch
  - Different data structures
    - Structured, unstructured, semi-structured
  - Different data paradigms
    - Relational, Dimensional, Document, Graph, Object-oriented, etc.
  - Different data types
    - JSON doesn’t have a date/time/duration type, XML schema and SQL have a variety, etc.
  - Different markup standards
    - XML, JSON, RDF, etc.
The Relational Model of Data for Large Shared Data Banks

E. F. CODD
IBM Research Laboratory, San Jose, California

Information Retrieval, Volume 13 / Number 6 / June, 1970

Programs should remain unaffected when the internal representation of data is changed. Tree-structured inadequacies are discussed. Relations are discussed and applied to the problems of redundancy and consistency.

KEY WORDS AND PHRASES: data base, data structure, data organization, hierarchies of data, networks of data, relations

CR CATEGORIES: 3.70, 3.73, 3.75, 4.20, 4.22

1. Relational Model and Normal Form

1.1. INTRODUCTION
This paper is concerned with the application of elementary relation theory to formatted data. The problems are those of data independence and data inconsistency.

The relational view appears to be superior in several respects to the graph or network model.

Relational view forms a sound basis for treating derivability, redundancy, and consistency. [and] a clearer evaluation of...

1.2. DATA DEPENDENCIES IN PRESENT SYSTEMS

...Tables represent a major advance toward the goal of data independence...

1.2.1. Ordering Dependence. Programs which take advantage of the stored ordering of a file are likely to fail if it becomes necessary to replace that ordering by a different one.

1.2.2. Indexing Dependence. Can application programs remain invariant as indices come and go?

1.2.3. Access Path Dependence. Many of the existing formatted data systems provide users with tree-structured files or slightly more general network models of the data. These programs fail when a change in structure becomes necessary. The program is required to exploit paths to the data. Programs become dependent on the continued existence of the paths.
How do variability, variety, and relevance map to the five data paradigms?
Five Data Paradigms

SQL DBs currently support 2 of 5 data models

✅ Relational Model
   - Data entry apps

✅ Dimensional Model
   - Data warehouse apps

❌ Object Model
   - Data processing apps

❌ Document Model
   - Document delivery apps

❌ Graph Model
   - Relationship apps
Relational Model

**Focuses on data**

- Manages data *entry*
- Data *freed from context*
- Data reuse, durability, integrity, independence, unity

- Formal data structures & truth statements
- Truth queries through dynamic, flexible SQL language

**Poor Variability:** Relationships not dynamic

**Poor Variety:**
- Impedance mismatch with OO
- Not designed for unstructured data

**Poor Relevance:** Queries cannot be sorted by relevance because they don’t have document context
Relational Model

Use for maximum flexibility in querying operational data

Example: traditional data-entry apps
Focuses on *information*

- Manages info *retrieval*
- Info *shown in context*
- Info reuse, durability, integrity, independence, unity

- Authoritative source, authoritative view, defined business questions
- Ad hoc self-service queries through fast & flexible semantic layer

**Poor Variability:** ✗ Unexpected changes break ETL jobs

**OK Variety:** ✓ Data can be extracted and loaded from *any* data source

**Good Relevance:** ✓ Queries return the most relevant data because information context is carefully defined
Dimensional Model

Use to transform authoritative data into contextual information to enable self-service, ad hoc, flexible reporting

Example:
traditional BI
Object Model

**Focuses on process**

- **✓** Manages data *processing*
- **✓** Data *processed in context*
- **✓** *Process* reuse, durability, integrity, independence, unity
- **✓** Formal processes through methods
- **✓** Flexible object queries through JPQL

**Best Variability:**

- **✓** JPA combined with OO DB provide query flexibility
- **✓** Agile development through object schema evolution, object links, embedded objects, class hierarchies
- **✓** Fast development because no language/model mismatch

**OK Variety:**

- **✓** No impedance mismatch: OO data is stored directly
- **✗** Object-oriented model not designed for unstructured data

**Poor Relevance:**

- **✗** Information hiding prevents querying text in context
- **✗** Objects cannot freely intermingle with text
Object Model

Use to manage complex processes and to speed & simplify application development

Example:
traditional data-processing apps
**Document Model**

**Focuses on content**
- ✓ Manages documents
- ✓ Content processed in context
- ✓ Document reuse, durability, integrity, independence, unity

- ✓ Informal document structure *(hierarchical & semantic)*
- ✓ Flexible & accurate document processing through queries *(recall)*
- ✓ Relevance queries through composable, customizable search *(precision)*

<table>
<thead>
<tr>
<th>Hospital Name</th>
<th>John Hopkins</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation Number</td>
<td>13</td>
</tr>
<tr>
<td>Operation Type</td>
<td>Heart Transplant</td>
</tr>
<tr>
<td>Surgeon Name</td>
<td>Dorothy Oz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Drug Name</th>
<th>Drug Manufacturer</th>
<th>Dose</th>
<th>Dose UOM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minicillan</td>
<td>Drugs R Us</td>
<td>200</td>
<td>mg</td>
</tr>
<tr>
<td>Maxicillan</td>
<td>Canada4Less Drugs</td>
<td>400</td>
<td>mg</td>
</tr>
<tr>
<td>Minicillan</td>
<td>Drug USA</td>
<td>150</td>
<td>mg</td>
</tr>
</tbody>
</table>

Best Variability: ✓ Content is dynamic and drives all processing

Good Variety: ✓ Fast development because very little impedance mismatch
- ❌ Requires defensive programming to handle variant data

Good Relevance: ✓ All content is in context so relevance is possible
- ✓ New data & relationships can be discovered and marked up
XML vs. JSON

This paper is....

The <i>relational</i> model is no longer, the only game in town.

XML is more expressive and more complex than JSON
1. It allows text to be freely interspersed among elements
2. It requires every element to be part of a namespace
3. It allows elements to be precisely typed (XML schema)
4. It is harder to read, parse, and its markup consumes more space
Use **XML** documents to enable search relevance

Use **JSON** documents to enable agile development

**Example:**

*new browser-centric apps*
Graph Model

**Focuses on relationships**

- Manages relationships
- Relationships create context
- *Relationship* reuse, durability, integrity, independence, unity
- Informal *relationship* structure
- Relationship queries
- Flexible queries through pattern matching
- Data is so granular it is difficult to comprehend & manage

Good Variability: ✓ Everything is dynamic: maximum flexibility

Good Variety: ✓ Fast development because very little impedance mismatch
- Requires defensive programming to handle variant data

Good Relevance: ✗ Relationship patterns create richer relevance than text
- Original text and context is lost
Graph Model

Use to enhance text search using relationships and relationship patterns

Example: genetic research and map apps
# Modeling Takeaways

**Each model has a specialized purpose**

<table>
<thead>
<tr>
<th>Model</th>
<th>Optimal Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensional</td>
<td>Self-service ad hoc reporting</td>
</tr>
<tr>
<td>Relational</td>
<td>Flexible queries &amp; joins</td>
</tr>
<tr>
<td>Object</td>
<td>Processes &amp; agile programming</td>
</tr>
<tr>
<td>Doc</td>
<td>Search &amp; transformations</td>
</tr>
<tr>
<td>Graph</td>
<td>Relationship relevance &amp; patterns</td>
</tr>
</tbody>
</table>

- Use the model that best fits your needs
- Use multiple models in the same app
- Glue it all together
What NoSQL model is best for developing apps?

**Document model**

- Increases developer productivity by 2x over relational – if you don’t need to compensate for lack of consistency and data integration
- Supports agile development without a schema
- Handles rapidly changing requirements
- Handles deeply hierarchical, complex, and highly variable data structures
- Has little-to-no impedance mismatch between application and database
- JSON is the new lingua franca of the web
- Potential to enable better search relevance
  - Full-text search in context of document structure
  - Full-featured queries of any data anywhere in a document
What does an App need from a DB?

- Insert, Update, and Delete data
- Query data
- Search data
- Bulk Process data
- Do it all consistently
## Which document NoSQL DBs are best for developer productivity?

<table>
<thead>
<tr>
<th>Feature</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low barrier to entry</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Poor</td>
</tr>
<tr>
<td>Easy to insert, update, delete individual documents</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
</tr>
<tr>
<td>Easy to retrieve one document</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
</tr>
<tr>
<td>Query to find multiple docs</td>
<td>Great</td>
<td>Good</td>
<td>Poor</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Search within documents</td>
<td>Great</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Search relevance</td>
<td>Great</td>
<td>Poor</td>
<td>Good</td>
<td>Poor</td>
<td>None</td>
</tr>
<tr>
<td>Advanced search : Facets, Geospatial, Entity Enrichment</td>
<td>Great</td>
<td>None</td>
<td>None</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Graph capabilities</td>
<td>Poor</td>
<td>None</td>
<td>Poor</td>
<td>None</td>
<td>None</td>
</tr>
<tr>
<td>Doc bulk-processing</td>
<td>Good</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>None</td>
</tr>
<tr>
<td>Data Integration Capabilities</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Data Consistency</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
</tbody>
</table>
Agenda

• What is NoSQL?
  – New Modeling Paradigms
  – New Hardware Paradigms

• Which NoSQL is best?
New Database Paradigms

Graph
- AllegroGraph, Neo4j, InfiniteGraph
- OrientDB
- DEX, SysTap

Document
- MarkLogic, MongoDB, CouchDB, CouchBase

Column
- JSON
- key/val
- DynamoDB, Riak, Cassandra, Hbase, HyperTable, Oracle NoSQL

newSQL
- Oracle x10, VoltDB, Clustrix

Live Analytics
- Oracle Exalytics, SAP HANA

Big Data
- Hadoop
- JSON
- XML

Doc Warehouse
- MarkLogic, Solr
- XML

Versant, Objectivity, GemStone

Object

SQL
- Oracle DB, SQL Server, DB2, Sybase, MySQL, PostgreSQL

Data Warehouse
- Exadata, OLAP, Teradata, Netezza, Sybase, Greenplum, Vertica, ParAccel

Raw

Less structure (schemaless)

High Bandwidth Analytical Volume

Operational Velocity

Low Latency

MarkLogic, Solr

Raw

Hadoop

Hadoop

XML

XML

More structure (schema)
Do we need to optimize databases differently for velocity and volume?
## Storage Cost/Performance

Determine Physical Architecture

<table>
<thead>
<tr>
<th>Optimization</th>
<th>Technology</th>
<th>Persistent</th>
<th>Capacity</th>
<th>Inexpensive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Volume</strong></td>
<td>Hard Drives</td>
<td>Y</td>
<td>High</td>
<td>Capacity, Bandwidth</td>
</tr>
<tr>
<td><strong>Velocity</strong></td>
<td>RAM</td>
<td>N</td>
<td>Low</td>
<td>IOPs, Latency, Bandwidth</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Technology</th>
<th>GB / Blade</th>
<th>Latency (µS)</th>
<th>Bandwidth (MB/s)</th>
<th>IOPs (1000/s)</th>
<th>Cost / GB</th>
<th>Cost / MB/s</th>
<th>Cost / IOPs</th>
<th>Cost / Blade</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAID 0 w/2 HDDs</td>
<td>1,200</td>
<td>4,850</td>
<td>288</td>
<td>2.4</td>
<td>1.25</td>
<td>5.21</td>
<td>0.63</td>
<td>1.5K</td>
</tr>
<tr>
<td>Flash SSDs</td>
<td>800</td>
<td>123</td>
<td>1,308</td>
<td>29</td>
<td>31.25</td>
<td>19.12</td>
<td>0.86</td>
<td>25.0K</td>
</tr>
<tr>
<td>Flash PCIe SLC</td>
<td>320</td>
<td>26</td>
<td>1,200</td>
<td>238</td>
<td>53.13</td>
<td>14.17</td>
<td>0.07</td>
<td>17.0K</td>
</tr>
<tr>
<td>RAM DDR3-1333 DR (12x16GB)</td>
<td>192</td>
<td>0.03</td>
<td>5,333</td>
<td>666,500</td>
<td>65.10</td>
<td>2.34</td>
<td>0.00</td>
<td>12.5K</td>
</tr>
<tr>
<td>RAM DDR3-1333 SR (10x4GB)</td>
<td>40</td>
<td>0.02</td>
<td>10,666</td>
<td>1,333,000</td>
<td>33.43</td>
<td>0.13</td>
<td>0.00</td>
<td>1.3K</td>
</tr>
</tbody>
</table>
Hardware Architecture

Velocity Optimized (OLTP)

- JSON
- XML

MPP RAM DB

Sync or Async

Volume Optimized (Warehouse)

- JSON
- XML

MPP Disk DB

Sync or Async

Sync or Async Transactions
What’s wrong with SQL DB’s?

- **Velocity**
  - SQL DBs are *serialized* to ensure consistency and use high latency disk, and this prevents them from scaling horizontally.

- **Volume**
  - Because SQL DBs *share* cores, caches, and storage, they are *serialized* across these resources, and this prevents them from scaling horizontally.
Problem: **Serialized DB Design**

DBs use serialization and synchronization for consistency

- Loading data from disk to RAM
- Data Processing 4%
- Recovery 24%
- Locking rows of data
- Locking buffered data
- Buffer Pool 24%
- Latching 24%
- Locking 24%

Michael Stonebraker @ 2011 NoSQL Conference San Jose
Cost of Synchronization

Single threaded operations are **746 times faster** than multiple threads with locks

**Time (ns)**

- Single thread: 300
- Single thread with memory barrier: 4,700
- Single thread with CAS: 5,700
- Single thread with lock: 10,000
- Two threads with CAS: 30,000
- Two threads with lock: 224,000

*Slide from James Gates SORT presentation, “When Performance Really Matters”*
Future is More Cores — Not Faster

- Exponential Transistors
- Flat Clock
  - (faster clock needs more power)
- Flat Power
  - (power is expensive and hot)
- Flat ILP
  - (instruction-level parallelism)
Hardware Takeaways

Velocity and Volume

- Future architectures will scale horizontally (MPP)
- They will be a mix of synchronous and asynchronous transactions
- RAM is lowest cost for velocity (low latency)
- Disk is lowest cost for volume (high bandwidth)
- MPP functional processing is needed to glue it all together (input → function → output)
## Which document NoSQL DBs perform best for apps?

<table>
<thead>
<tr>
<th>App Performance</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Couch...</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Insert Velocity</td>
<td>Good</td>
<td>Good</td>
<td>Great</td>
<td>Good</td>
<td>Great</td>
</tr>
<tr>
<td>Volume Scalability</td>
<td>Great</td>
<td>Good</td>
<td>Great</td>
<td>Good</td>
<td>Great</td>
</tr>
<tr>
<td>Fast retrieval of one doc</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
</tr>
<tr>
<td>Fast query of docs (secondary index)</td>
<td>Great</td>
<td>Great</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Fast query of docs (no secondary index)</td>
<td>Great</td>
<td>Good</td>
<td>Poor</td>
<td>Good</td>
<td>None</td>
</tr>
<tr>
<td>Fast search within documents</td>
<td>Great</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Fast search relevance</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>None</td>
</tr>
<tr>
<td>Fast aggregation</td>
<td>Great</td>
<td>Good</td>
<td>Poor</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Fast map-reduce</td>
<td>Poor</td>
<td>Good</td>
<td>Great</td>
<td>Great</td>
<td>None</td>
</tr>
<tr>
<td>Hadoop integration</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
<td>Great</td>
</tr>
</tbody>
</table>
Agenda

• What is NoSQL?
• Which NoSQL is best?
  – Consistency Model
  – Enterprise Readiness
What’s wrong with NoSQL?

• In most NoSQL solutions, the developer is responsible for ensuring consistency
  – Imagine programming an app to coordinate thousands of concurrent threads across gigabytes of data structures
  – Imagine writing code to handle all threading issues
    • Locks
    • Contention
    • Serialization
    • Dead locks
    • Race conditions
    • Threading bugs
    • Etc
What are ACID transactions?

Databases use the ACID model to make it easy, reliable, and fast for hundreds of concurrent processes to query and modify the same data consistently:

Contrast this with manually programming hundreds of threads to do the same

- **Atomic**
  - All parts of a transaction succeed, or all fail and roll back

- **Consistent**
  - All committed data must be consistent with all data rules including constraints, triggers, cascades, atomicity, isolation, and durability

- **Isolated**
  - No transaction can interfere with other concurrent transactions

- **Durable**
  - Once a transaction is committed, data will survive system failures, and can be recovered after an unwanted deletion
What is Durability?

Durable data survives system failures

- Once a transaction has committed, its data is guaranteed to survive system failures
  - Failures may occur in the server, operating system, disk, and database.
  - Failures may be caused by server crash, full disk, corrupted disk, power outages, etc.
- Durability requires storage to be **redundant**.
- Durability requires **logs to replay** asynchronously written data.
- Durability requires **logs to be archived** to another location so they can be recovered.
- Durability works with **atomicity** to ensure that partially written data is not durable.
- **Without durability** you can have faster inserts, updates, and deletes because you have no logs to write and you can store data in volatile memory while lazily writing it to disk.

Durable data can be recovered after unwanted deletion

- Durability allows data to be recovered to a point before the system failed or before applications or users inappropriately destroyed or modified data.
- Durability requires backups of data.
- Backups need to be able to recover data to a point in time.
Do you need Durability?

• Can you live with writing **advanced code** to compensate?
  – Are you willing to trust all developers to properly check for partial transaction failures, properly detect current physical layout of the database cluster, and write extra code to properly propagate data durably across the cluster?

• Can you live with **lost data**?
  – Can you live with data lost because the database doesn’t use logs, doesn’t archive it logs, doesn’t use mirrored disks, and doesn’t require data to be written to multiple servers?

• Can you live with **accidental deletion of data**?
  – Can you live without a point-in-time recovery feature that can restore data immediately before an accidental deletion?

• Can you live with **scripting your own** backup & recovery solution?
  – Are you willing to develop custom scripts to back up and restore data files and database configurations?
# Durability Support

<table>
<thead>
<tr>
<th>Durability — Data Survival</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Durable on commit</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Recoverable to point-in-time</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Recoverable from backup</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

## NOTES

- MongoDB’s architecture is fundamentally not durable. By default it is not durable. Configuring MongoDB to be durable is complicated and error-prone. Making MongoDB durable severely hurts performance. Even when properly configured to be durable, its in-memory design allows data to be committed without being persisted.

- To ensure durability in Riak and Cassandra, each transaction must commit across two or more nodes. If only one node reports success, your transaction is not durable; and if that node fails, your data is lost. When this happens in Riak, you can immediately read from the desired nodes to “repair” the failed writes. In Cassandra, you have to manually retry your inserts. In addition, in Riak and Cassandra the document distribution algorithm does not guarantee that documents will go to different physical nodes, which means there is no guaranteed durability.
What is Atomicity?

An atomic transaction is *all or nothing*

- Atomicity means all parts of a transaction succeed or nothing does.
- This requires partially written data to be automatically removed from disk.
- A single command or a set of commands is called a transaction.
- A single command to a programmer often represents multiple commands to the database because the database needs to replicate data to multiple disks, to update indexes, to execute triggers, to verify constraints, and to cascade deletes and updates, etc.
- **Without atomicity**, you can have faster transactions because they don’t need a two-phase commit.

**Sets of data**

- An operation may need to process multiple data items.
- All data in the set needs to be changed or none; otherwise it becomes *arbitrarily inconsistent*.
- For example, you want to **delete** part of a set of data and part way through the transaction fails. If all the changes are not automatically rolled back, the data is left in an inconsistent state that can affect the results of other transactions.

**Sets of commands**

- A series of commands often needs to work as a unit to produce accurate results.
- All commands need to succeed or all need to fail; otherwise data becomes *arbitrarily inconsistent*.
- Inconsistent data is hard to fix: data may contradict other data, and there may be **extra data** or **missing data**. Without atomicity to roll back failures, there may be no way to fix the data.
- The classic example is where you need to debit one account and credit another as a single transaction. If one command succeeds and the other fails, account totals are inaccurate.
Do you need Atomicity?

• Can you live with **modifying single documents at a time**?
  – Without atomicity, you can’t guarantee results when working with multiple documents at a time

• Can you live with **partially successful transactions**?
  – Without atomicity, you can have higher availability because transactions can partially succeed
  – For example, rather than losing a sale when an app failed to capture a credit card number, you can save what you captured and later contact the customer to get the rest.

• Can you live with **inconsistent and incomplete data**?
  – Without atomicity, sets of data or sets of commands may fail before being processed completely leaving data inconsistent or incomplete
  – Is it OK not to know when data anomalies are caused by bugs in your code or are temporarily inconsistent because they haven’t yet been synchronized?

• Can you live with writing **advanced code** to compensate?
  – Are you willing to develop custom solutions to provide atomic rollback, such as transaction logs, version numbers, and two-phase commits?
  – Are you willing to program defensively to handle transactions that fail?
  – Are you willing to develop processes to find and fix inconsistent data?
# Atomicity Support

<table>
<thead>
<tr>
<th>Atomicity — All or nothing commit</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>For one item plus indexes</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>For one item plus validation rules</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Across a set of data</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Across a set of commands</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Across a set of nodes</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>

**NOTES**

- In Riak and Cassandra to ensure durability, each transaction must commit across two or more nodes. If only one node reports success your transaction is not durable — yet these DBs do not roll back the transaction. Thus, if you want durability, they cannot provide atomicity for an insert, update, and delete of a *single document*. To make matters worse, in Riak and Cassandra, if you need to index data in multiple ways, you have to insert the same document multiple times with different keys. Some of those inserts may succeed and some might not. Thus, Riak and Cassandra cannot provide any level of atomicity.
What is Isolation?

Isolation prevents concurrent transactions from affecting each other

- **Read isolation** ensures queries return accurate results as of a point in time
- **Write isolation** locks data to prevent race conditions during updates, deletes, and inserts
- Without isolation, queries and transactions run faster because the database doesn’t have to provide a consistent view using locks, snapshots, or system version numbers

**• Sets of data**

- An operation can only produce accurate results as of a **point in time**
  - It takes time for a command to process a set of data
  - During this time, concurrent transactions may insert, update, and delete data in the set
  - Without isolation, a single command executes against data while it is being changed by other concurrent transactions.
  - Records may be added after the command started running
  - Records may be deleted or changed after the command has processed them.
  - This creates inconsistent results: aggregate functions produce wrong answers

**• Sets of commands**

- A series of commands need to work on a consistent view of data to produce accurate results
  - Without isolation, each command in a series will execute against arbitrarily different data
Do you need Isolation?

- Can you live with never processing more than a single document?
  - Without isolation, you can’t guarantee results if you work with multiple documents at a time.

- Can you live with inaccurate queries?
  - Without isolation, query results are inaccurate because concurrent transactions change data while processing it.

- Can you live with race conditions and deadlocks?
  - Without isolation, transactions can create race conditions and deadlocks.

- Can you live with writing advanced code to compensate?
  - Are you willing to implement your own versioning system (like MVCC) that provides a reliable system change number spanning multiple data centers?
  - Are you willing to write code to hide concurrent updates, inserts and deletes from queries?
  - Are you willing to write code to handle update conflicts, race conditions and deadlocks?
# Read Isolation Support

<table>
<thead>
<tr>
<th>Isolation — read non-interference</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>MVCC</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

## OLTP Query

| — for one item                         | Yes       | Yes     | Yes   | Yes              | Yes       |
| — for multiple items                   | Yes       | No      | No    | Yes              | No        |
| — for multiple commands                | Yes       | No      | No    | No               | No        |

## Batch Queries (map/reduce)

| — for multiple items                   | Yes       | No      | Yes   | Yes              | No        |
| — for multiple commands                | Yes       | No      | No    | No               | No        |
# Write Isolation Support

<table>
<thead>
<tr>
<th>Isolation</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Write non-interference</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Update</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—for one item (optimistic – no lock)</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>—for one item (pessimistic – lock)</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>—for multiple items</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>—for multiple commands</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Delete</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—for one item</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>—for multiple items</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>—for multiple commands</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Insert</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>—for one item</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>—for multiple items</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>—for multiple commands</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
</tbody>
</table>
What is Consistency?

- All committed data must be consistent with all data rules
  - Constraints, triggers, cascades, atomicity, isolation, and durability

- Data must always be in a consistent state at any point in time

- Consistency is the product of atomicity, isolation, and durability
  - Atomicity ensures that if data rules are violated, such as constraints and triggers, the transaction fails and all changes are rolled back.
  - Isolation ensures a query sees a consistent set of data while other concurrent commands are modifying the underlying data.
  - Isolation ensures bulk updates lock sets of data so they can be processed as a consistent unit without other concurrent commands modifying their data.
  - Durability ensures that data is consistently replicated to other nodes in a cluster so a loss of a node won’t cause a loss of data.
Do you need Consistency?

Not necessarily — instead, you may prefer

- Absolute fastest performance at lowest hardware cost
- Highest global data availability at lowest hardware cost
- Working with one document at a time
- Writing advanced code to create your own consistency model
- Eventually consistent data
- Some inconsistent data that can’t be reconciled
- Some missing data that can’t be recovered
- Some inconsistent query results
## Consistency Support

<table>
<thead>
<tr>
<th>Consistency — results are accurate and repeatable at a point in time</th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>True consistency</td>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Eventual consistency</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-datacenter clusters</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Multi-master replication</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

**NOTES**

- Eventual consistency is an oxymoron: it means your data and queries may be or may not be consistent: this is fundamentally inconsistent.
- Multi-master replication in all NoSQL databases requires the application to handle conflict resolution. This is a non-trivial problem to solve.
- MarkLogic does not support multi-master replication because it is fundamentally inconsistent and thus incompatible with the ACID consistency model.
What do you need most?

– Highest performance for queries and transactions

– Highest data availability across multiple data centers

or

– Less data loss (i.e. durability)

– More query accuracy & less deadlocks (i.e. isolation)

– More data integrity (i.e. atomicity)

– Less code to compensate for lack of ACID compliance
Compensating for Cassandra

“Hystrix is a library designed to control the interactions between these distributed services providing greater tolerance of latency and failure. Hystrix does this by isolating...

...a dramatic improvement in uptime and resilience has been achieved through its use”
Compensating for Cassandra

Announcing Blitz4j - a scalable logging framework

By Karthikeyan Ranganathan

We are proud to announce Blitz4j, a critical component of the Netflix logging infrastructure that helps Netflix achieve high volume logging without affecting scalability of the applications.

“Blitz4j is a logging framework...to reduce multithreaded contention and enable highly scalable logging...”

Netflix is one of the largest Apache Cassandra production deployments in the world. We've accomplished this by having extremely talented engineers building tools to support Cassandra and contribute features and code to the Apache Cassandra project itself. All of this while hosting it in the Cloud!
Agenda

• What is NoSQL?
• Which NoSQL is best?
  – Consistency Model
  – Enterprise Readiness
Which document NoSQL DBs are ready for the enterprise?

<table>
<thead>
<tr>
<th></th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant CouchDB</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cost</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>Data Integration</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Security</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
<tr>
<td>Administration: monitoring, provisioning, patching, etc.</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
</tbody>
</table>

NOTES

- NoSQL is as expensive as SQL databases, such as Oracle and SQL Server.
- Hardware costs (particularly storage) are the most expensive and comparable.
- Open source databases have no licensing costs, but they have annual maintenance.
- Vendor’s annual maintenance costs are comparable.
When will NoSQL be Enterprise Ready?

- In-memory DB
- MapReduce
- DB as a Service
- Open Source DBs
- Columnar DW
- DB Appliances
- Oracle SQL Server DB2

Technology Trigger → Inflated Expectations → Disillusionment → Enlightenment → Productivity

- Enterprise Ready
- 1 to 5 years
- 5 to 10 years

Derived from Gartner Hype Cycle for Data Management
What document NoSQL database is best?

<table>
<thead>
<tr>
<th></th>
<th>MarkLogic</th>
<th>MongoDB</th>
<th>Riak</th>
<th>Cloudant</th>
<th>Cassandra</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall Developer Productivity</td>
<td>Great</td>
<td>Good</td>
<td>Poor</td>
<td>Good</td>
<td>Poor</td>
</tr>
<tr>
<td>Overall App Performance</td>
<td>Great</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
<td>Good</td>
</tr>
<tr>
<td>Overall Enterprise Readiness</td>
<td>Great</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
<td>Poor</td>
</tr>
</tbody>
</table>

Questions / Comments?