Database Systems

April. 24, 2015

Overview

Relational Database

NoSQL Databases

Need For DBMS

How to store and process data **in memory**?

- By data structures ({sequence, associative} container) and their interfaces (insert, lookup and delete)

How to store and process data **on disk**? (Their differences: volatility, random-access, etc)

- Option 1: write data structures to **files** and read files to memory.
  - Good enough for large objects (image, video), bad for **structured** data. (think about an array of struct)
    - The type, size information of the struct also needs to be stored in file.
    - We should avoid storing a single struct across disk block boundaries.
- Option 2: let another layer of abstraction to deal with those details.
  - Like all software/system/service/function do, DBMS simply provides an convenient interface and hides details.
  - *There are also **I**n-**M**emory **D**ata**b**ases (IMDB), mainly for performance and compatibility with DBMS on disks.

DBMS Interface

Create, **R**ead, **U**pdate, **D**elete (CRUD)

Relational Database

Data model:

- Relations (tables) = tuples (rows) * attributes (columns) (think about Excel)
- Relational algebra (set theory): rename, selection, projection, etc.

Interface:

- Structured Query Language (SQL), FROM, WHERE, SELECT, etc.

Features overview:

- Integrity constraints (type, primary key, unique, foreign key, etc.)
- Join
- Fast lookup with indexing
- Transactions
- ** Triggers, view, functions

Examples: MySQL, SQLite, PostgreSQL

CRUD

```
CREATE TABLE countries (
  country_code char(2) PRIMARY KEY,
  country_name text UNIQUE
);
```

```
INSERT INTO countries (country_code, country_name)
VALUES ('us','United States'), ('mx','Mexico'), ('au','Australia'),
       ('gb','United Kingdom'), ('de','Germany'), ('ll','Loompaland');
```

```
INSERT INTO countries
VALUES ('uk','United Kingdom');
```

```
DELETE FROM countries
WHERE country_code = 'll';
```

```
SELECT *
FROM countries;
```

```
 country_code | country_name
--------------+---------------
 us           | United States
 mx           | Mexico
 au           | Australia
 gb           | United Kingdom
 de           | Germany
 ll           | Loompaland
(6 rows)
```

```
CREATE TABLE cities (
  name text NOT NULL,
  postal_code varchar(9) CHECK (postal_code <> ''),
  country_code char(2) REFERENCES countries,
  PRIMARY KEY (country_code, postal_code)
);
```

```
INSERT INTO cities
VALUES ('Portland','87200','us');
```

```
INSERT INTO cities
VALUES ('Toronto','M4C1B5','ca');
```

```
UPDATE cities
SET postal_code = '97205'
WHERE name = 'Portland';
```

```
SELECT cities.*, country_name
FROM cities INNER JOIN countries
  ON cities.country_code = countries.country_code;
```

```
 country_code |   name   | postal_code | country_name
--------------+----------+-------------+---------------
 us           | Portland | 97205       | United States
```

Join

- Need for Join
  - Combine data from multiple tables.
  - Why separate them at the first place?
    - Flexible queries (Normalization)
    - Indeed stored together in NoSQL.
- Inner join
- Outer join
  - Left-outer join
  - Right-outer join
  - Full-outer join
- Natural join

Project	ID	Name	mgrssn
	1	Marketing	1
	2	Development	NULL
	3	Website	4

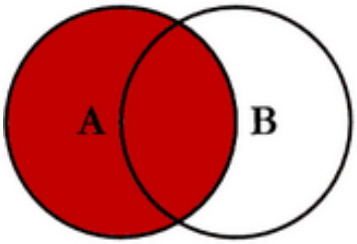
Employee	SSN	Firstname	Surname
	1	Alex	Jones
	2	Jamal	Al Hak
	3	Nissan	Imdana
	4	Johan	Fredriksson

InnerJoin	ID	Name	SSN	First	Surname
	1	Marketing	1	Alex	Jones
	3	Website	4	Johan	Fredriksson

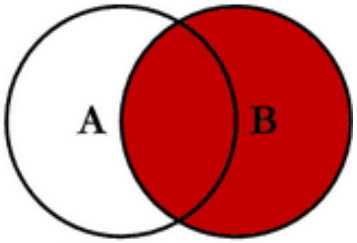
LeftOuter	ID	Name	SSN	First	Surname
	1	Marketing	1	Alex	Jones
	2	Development	NULL	NULL	NULL
	3	Website	4	Johan	Fredriksson

FullOuter	ID	Name	SSN	First	Surname
	1	Marketing	1	Alex	Jones
	2	Development	NULL	NULL	NULL
	3	Website	4	Johan	Fredriksson
	NULL	NULL	2	Jamal	Al Hak
	NULL	NULL	3	Nissan	Imdana

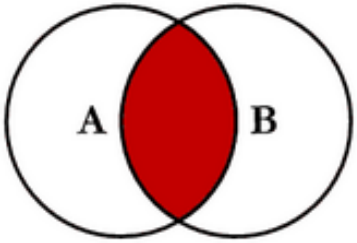
SQL JOINS



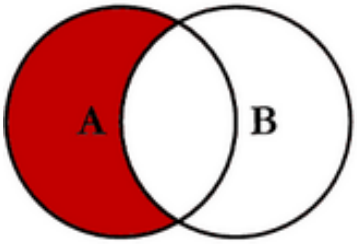

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
```



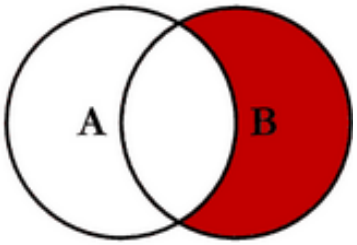

```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
```



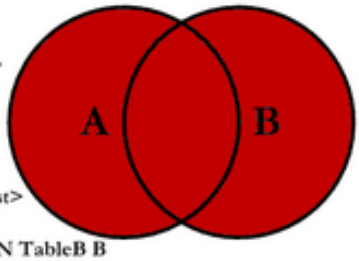

```
SELECT <select_list>
FROM TableA A
INNER JOIN TableB B
ON A.Key = B.Key
```



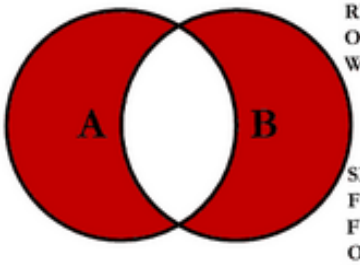

```
SELECT <select_list>
FROM TableA A
LEFT JOIN TableB B
ON A.Key = B.Key
WHERE B.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
RIGHT JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
```




```
SELECT <select_list>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.Key = B.Key
WHERE A.Key IS NULL
OR B.Key IS NULL
```

Exercise

https://leetcode.com/problems/combine-two-tables/

Indexing

Need for Indexing

Find all users whose annual income is between 50,000 and 55000.

Do we need to read all 1 billion records every time we process such query?

key-value pairs

key - one (or multiple) columns, value - pointer to the row on disk

Automatically created on the primary key and unique keys.

Unique constraint enforcement is done throughout those indices.

Manually CREATE INDEX

B-Tree index

Hash index

```
CREATE INDEX events_starts
  ON events USING btree (starts);

CREATE INDEX events_title
  ON events USING hash (title);
```

Transactions

Transaction: a **collection** of several database operations, a **single unit** from user's point of view.

Properties: **ACID**

- Atomicity: **all-or-nothing**.
  - Why: e.g. balance transfer, payment and seat reservation.
  - How: recovery system (logging).
- Consistency: data **consistent** (integrity constraints) is preserved after each transaction.
  - Why: prevent application errors.
  - How: integrity constraint checking.
- Isolation: **concurrent** execution of transactions results in a **system state** as if these transactions were executed **one at a time** in **some** order.
  - Why: prevent "lost update", "dirty read" problems
  - How: concurrency-control system (locking or versioning).
- Durability: once committed, changes **persist**, even in presence of system crash, etc.
  - How: Write to disk, recovery system.

Need for Isolation

Transaction concurrency problems

- “Lost Update”
  - transaction A: read record X; update address; write record X
  - transaction B: read record X; update income; write record X
- “Dirty Read”
  - Initial: balance X = 0, balance Y = 100;
  - transaction A: balance X + 100; balance Y - 100.
  - transaction B: withdraw 100 from X
  - transaction C: withdraw 100 from Y
- *More on http://goo.gl/mD14Uu

Scalability Issue

How to deal with increasing scale: more data, more incoming query-per-second?

- Use lots of machines (distributed database system)

Problems: **disk failures**, **network failures** are common

- Think about your wifi router…

Deal with disk failures: **data replication**

- Also spreads workload (improves simultaneous read performance)
- Not mandatory, but replicas are everywhere: local copy, cache

In presence of network failures, what should a database do?

CAP Theorem

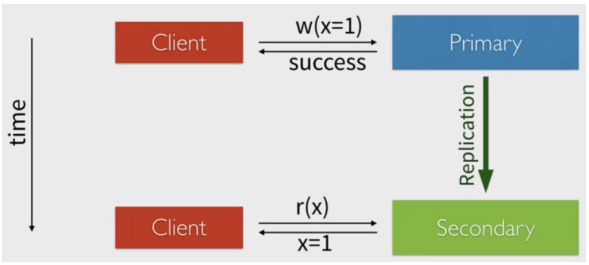
It's impossible for a distributed system to simultaneously guarantee all of:

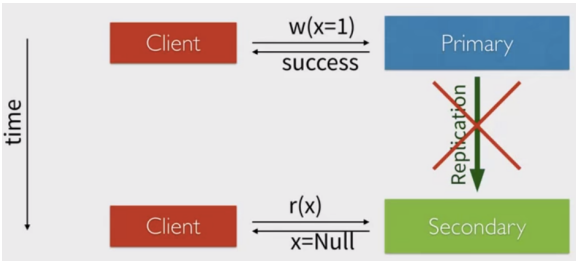
Consistency: all clients see the **same data** at the **same time**, as if there is only one copy.

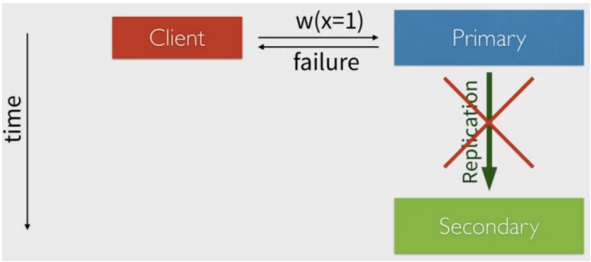
Availability: every transaction **eventually commits** or internally aborts

Partition tolerance: system continues to operate despite arbitrary message loss.

* Here the consistency problem is from multiple replicas (not multiple clients or multiple partitions)







More on CAP

The definition of P is vague: what is exactly “continue to operate”?

> You really cannot sacrifice P if you want a distributed system, simply because network failures are always there. Tradeoff is between **C and A**. [1, 2]

C in CAP is different from C in ACID

> C in CAP is a strict subset of ACID (where I is essential) [2]

CAP was initially a conjecture intended for justifying difficulties observed when creating a **perfect**-C and **perfect**-A distributed system. The later prove was debatably narrow.

> C and A are rather **continuous** than binary. [2] (elaborate on next slide)

Traditional RDBMS focuses on C (i.e. ACID properties) -> poor availability in distributed environment -> poor performance due to coordination.

NoSQL focuses on A (i.e. BASE properties: **B**asically **A**vailable, **S**oft state, **E**ventual consistency)

Eventual Consistency

N: the number of nodes a write will eventually (maybe asynchronously) propagate to.

W: min number of nodes that must *participate* in a successful write.

R: min number of nodes that must *participate* in a successful read. (compare version)

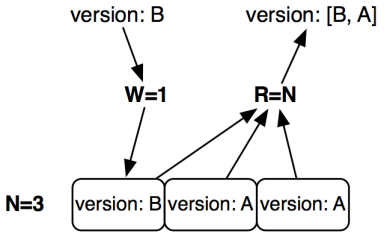


Figure 11—Consistency by reads: W=1, R=N

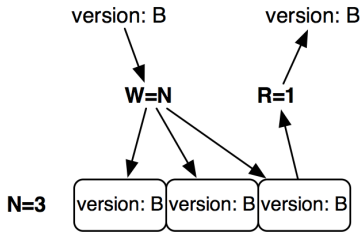


Figure 10—Consistency by writes: W=N, R=1

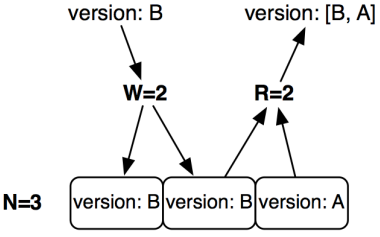


Figure 12—Consistency by quorum: W+R > N

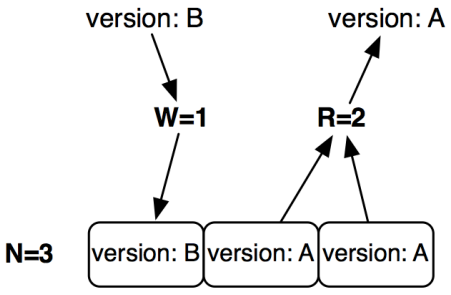


Figure 9—Eventual consistency: W+R <= N

Exercise

Does each of following systems prioritize C or A?

- Gmail
- Facebook status update
- Dropbox
- Amazon
- iMessage
- DNS
- Banking
- Ticket reservation system

When RDBMS?

Advantage:

- Battle-tested
- Versatile data modeling, flexible queries
- ACID-compliant transactions

Cases where it may not be a great fit:

- You need to **scale out** rather than scale up.
- You have **simpler** (more specific) data modeling and only want **restricted queribility** where relational schema is an overkill.
- You don't need the overhead of a full database functionality (**ACID**), but want high read/write throughput.

NoSQL Movement

Initially No-SQL, later interpreted as **N**ot-**O**nly-**SQL**.

Motivation:

RDBMS: one size fits all -> one size excels at nothing.

NoSQL: a wide range of specialized databases.

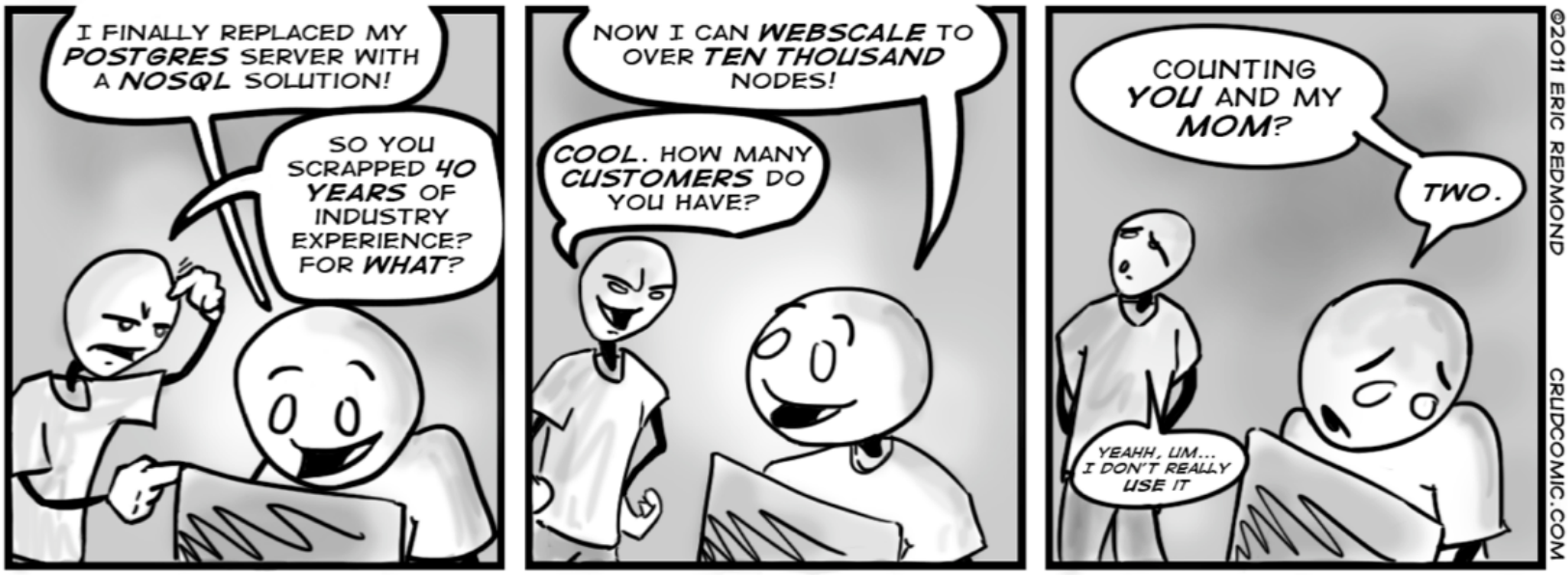
Goals: simplicity of design, horizontal scalability, finer tradeoff of C and A.

What has been changed? (a.k.a what are the causes of the RDBMS's poor scalability?)

SQL as an interface is **not** the problem. (Should be NoACID)

Strict consistency requirements (ACID) -> relaxed consistency models

Rigid relational schema -> flexible/specialized data models, for less flexible queries.


I FINALLY REPLACED MY POSTGRES SERVER WITH A NOSQL SOLUTION!
SO YOU SCRAPPED 40 YEARS OF INDUSTRY EXPERIENCE? FOR WHAT?
NOW I CAN WEBSCALE TO OVER TEN THOUSAND NODES!
COOL. HOW MANY CUSTOMERS DO YOU HAVE?
COUNTING YOU AND MY MOM?
TWO.
YEAHH, UM... I DON'T REALLY USE IT
©2011 ERIC REDMOND
CRUDCOMIC.COM


I DON'T UNDERSTAND THIS WHOLE REDIS THING.
WHY NOT JUST USE AN RDBMS WITH A TWO-COLUMN TABLE?
FOR THE SAME REASON YOU DON'T DRIVE A TANK TO WORK!
BECAUSE IT'S OUT OF MY PRICE-RANGE?
©2011 ERIC REDMOND
CRUDCOMIC.COM

Design Choices in NoSQL Databases

Detail design choices vary a lot.

- They choose to be specialized and different. (They only defines what it is not.)

Data model (How much database server know about the data structure?)

- Can be as simple as key-value pairs. (Think about in-memory map.)

Consistency level (C or A, ACID or BASE)

- A is more important for most web applications. (Think about memory consistency and cache coherency.)
- *How to solve “lost-update” problem in eventual consistency systems?
  - *Vector Clock Algorithm

Partitioning

Storage layout

Query API

Type of NoSQL Databases by Data Model

Key-Value Store: a map/dictionary/associative container on disk

Example: **Dynamo**, Riak, Redis, Memcached

Features: simplest data model (just above a plain file storing only values)

Column-oriented: a multidimensional map

Example: **BigTable**, HBase, Cassandra

Features: allow sparse columns

Document-oriented: document-id -> document (e.g. JSON)

Example: MongoDB, CouchDB

Features: Allow nested document; more flexible query (with more server-side knowledge of document structure).

…

Partitioning

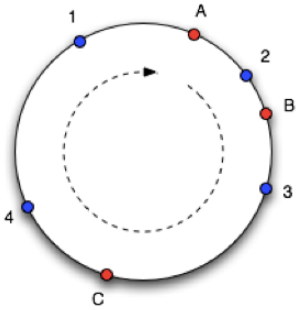
How to map from data partitions to storage nodes?

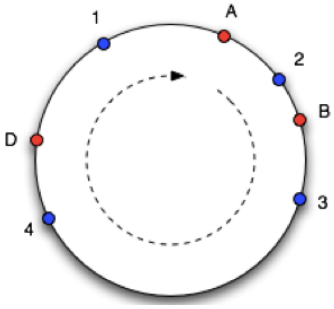
- hash(key) mod #nodes
  - Problem: nodes may join and leave at runtime (node crash, network partitioning, server maintenance). Hard to redistribute.
  - Solution: consistent hashing. (Dynamo)
- B-tree based range partitioning. (BigTable)

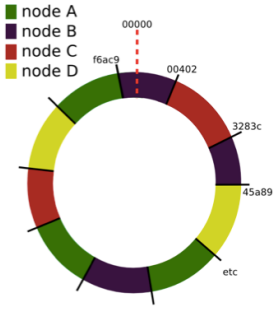
Join becomes very expensive (many network requests)

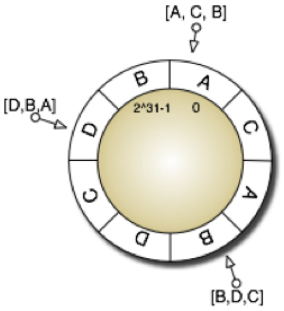
Consistent Hashing

Idea: Hash both keys and nodes using the same hash function, so that each machine gets a range of hashed keys. (Like range partitioning!)









Exercise: given a key, how to get the node it should store in.

Problem: If #nodes is small, unbalanced partition may be severe

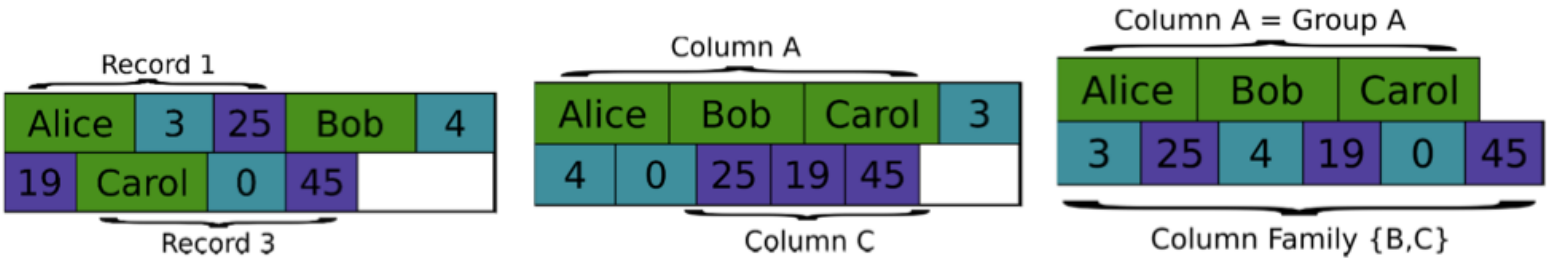
Solution: virtual nodes.

(Good) side effect: adjust load according to hardware resource.

Problem: data will be lost if a node leaves.

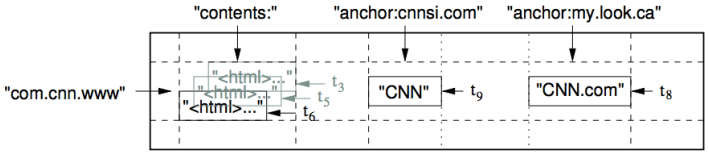
Solution: replicate.

Storage Layout



(a) Row-based **(b)** Columnar **(c)** Columnar with locality groups

BigTable



Data Model: (row-key, column-key, timestamp) -> value

timestamp: can be real or logical, yet another dimension.

column-key: “column-family:column-qualifier”.

column-family: a group of columns, unit of access control / replication.

Consistency:

Provides single-row transaction (even if split into multiple locality groups), no multi-row transaction.

Partition:

B-Tree like range-based partition.

Local storage:

Sorted by row-key. How to design row-key?

Efficient range query v.s. avoid hotspot (e.g. com.cnn.www)

Keep row size small v.s. keep row-key short (e.g. user1:201504)

Store (row-key, column key, timestamp) -> value in SSTable. (immutable file, with row index. Essentially a map on disk.)

Empty columns don’t take space -> allow for sparse column.

How to efficiently check whether a column **exists** for a given row? (get the row, iterate)

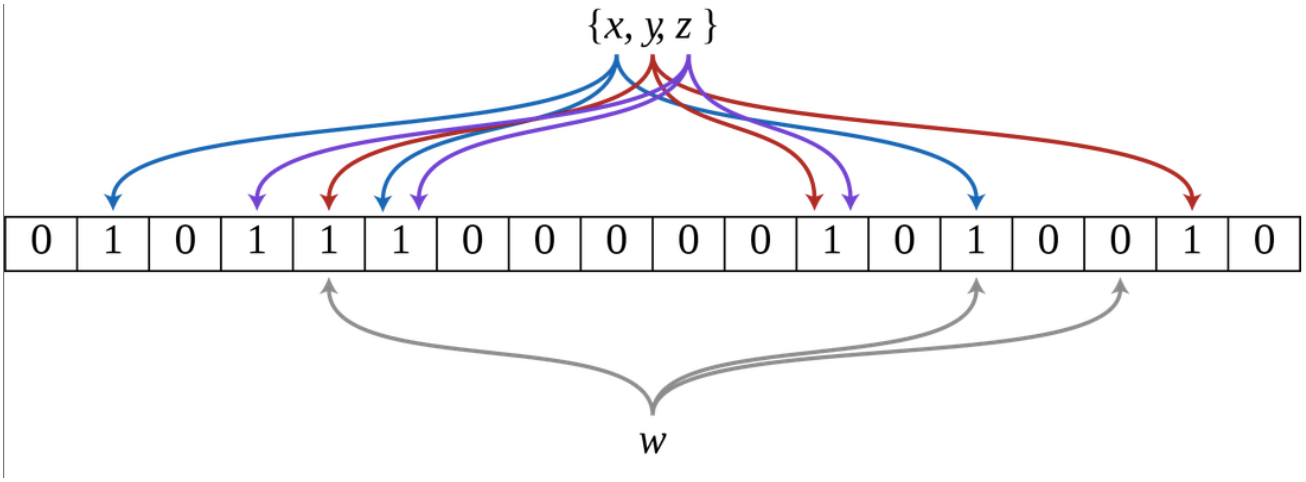
Bloom Filter. (User-specified on certain column-families)

User-specified locality group: a group of column-families. stored together, in memory…

Bloom Filter

Space-efficient bitmap. (for testing existence)

10 bits per element -> 1% false positive.



Dynamo

Data model: key -> value store (values are BLOBs)

Consistency: “write to shopping carts should always succeed.”

Versioning: vector lock, conflicts resolved by client-side read logic or “last-write-win”.

quorum-like replication consistency control: configurable N, W, R

Partition:

Consistent hashing with virtual node and replication.

Local storage:

Allow for plugin, Berkeley DB, MySQL

Cassandra

BigTable + Dynamo

Data Model: (row-key, column-key) -> value

- column-family, super-column-family

Consistency:

- Configurable N, W, R

Partitioning:

- Consistent hashing
- Provides “order-preserving hashing” mode -> unbalanced hash function
- Call for “active” load balancing.

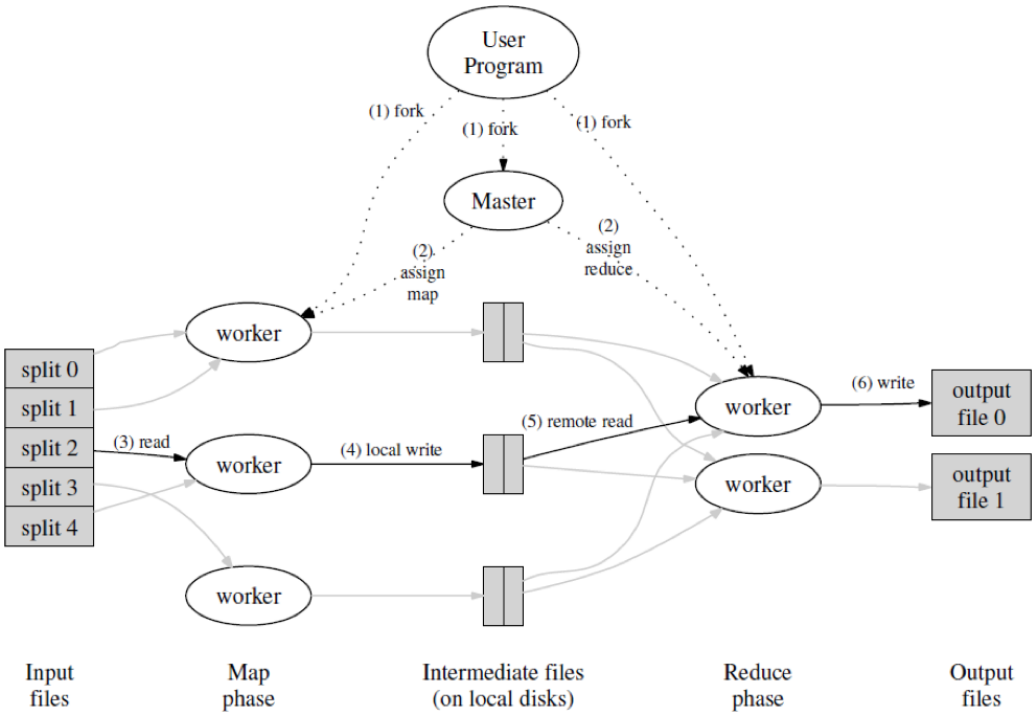
Local storage:

- File, with row index and bloom filter.

MapReduce

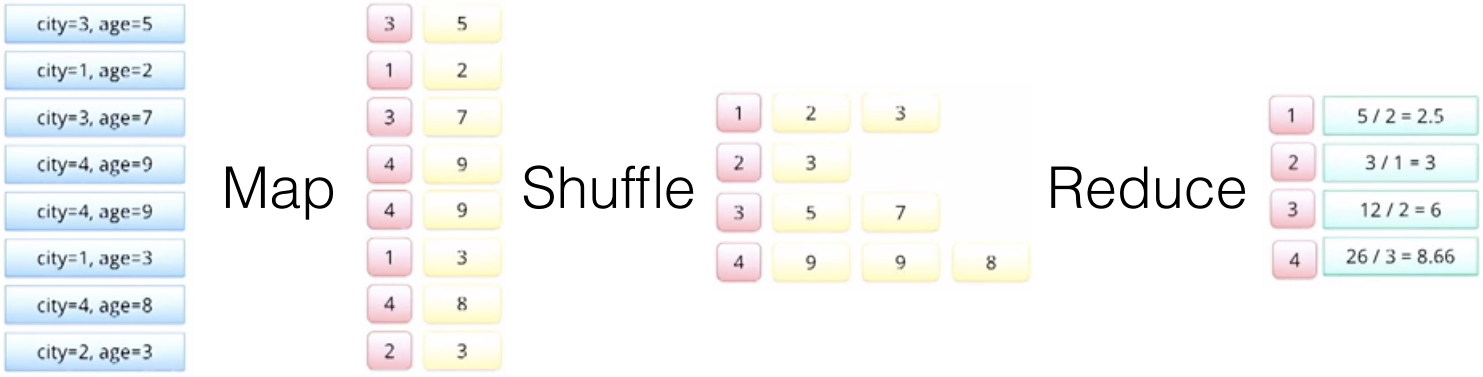
Need for MapReduce: (from NoSQL databases' perspective)

How to aggregate multiple rows? (like MAX, AVERAGE in SQL)



MapReduce Example

Compute average age of people in each city.



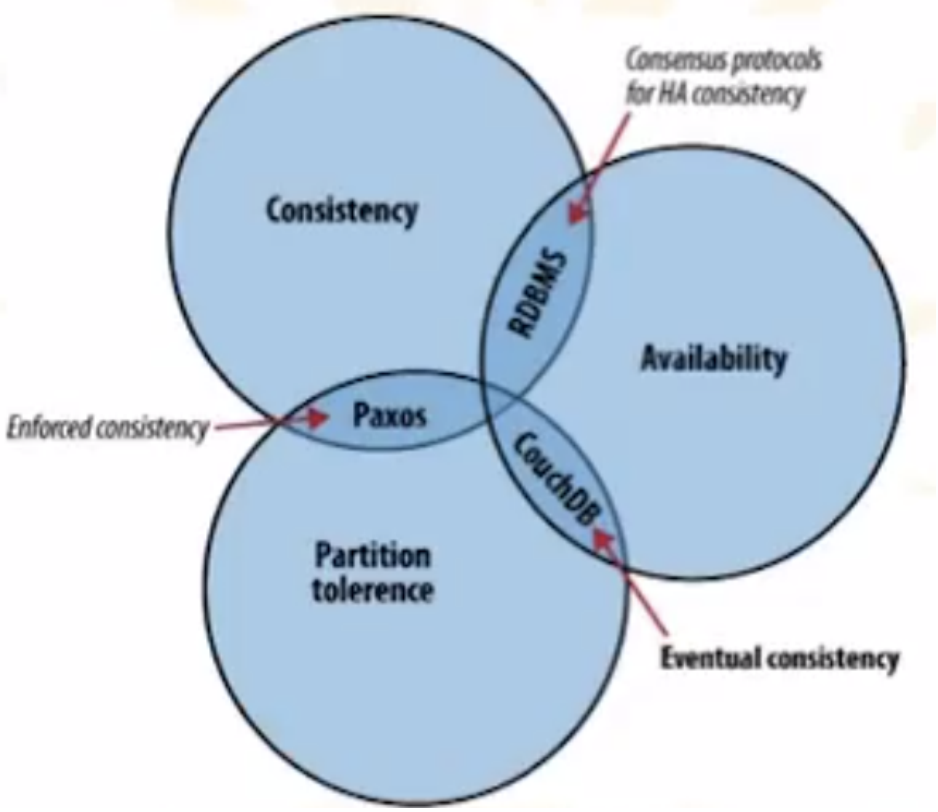
*NewSQL

The flaws of traditional RDBMS is not SQL, nor ACID, but it is old, not designed for shared-nothing architecture.

Paxos algorithm: The optimal algorithm for ensuring consistency between shard-nothing distributed environment.

Example: MIT H-store, Google Spanner

CAP tradeoff



References

[1] http://blog.cloudera.com/blog/2010/04/cap-confusion-problems-with-partition-tolerance/

[2] http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

[3] “Seven Databases in Seven Weeks - A Guide to Modern Databases and the NoSQL Movement” by Eric Redmond and Jim R. Wilson

[4] “NoSQL Databases” by Christof Strauch

[5] “Database System Concepts”