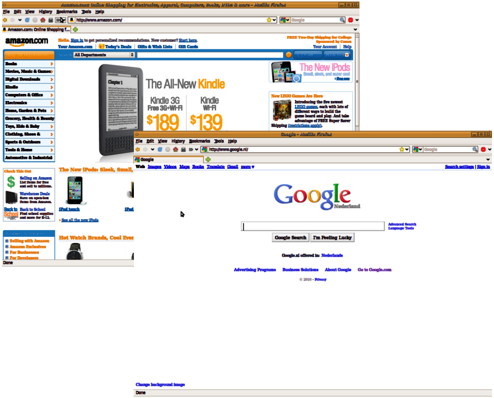
Développement logiciel pour le Cloud (TLC)

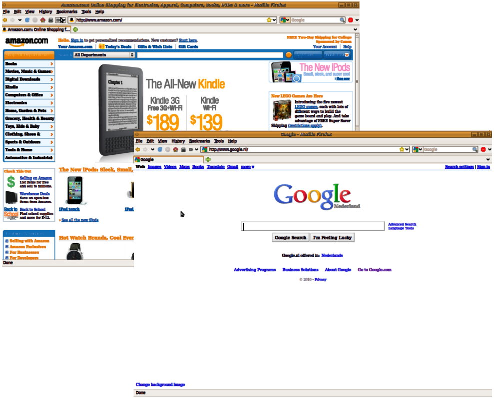
Introduction

Quentin Dufour

The Cloud is great for hosting Web applications

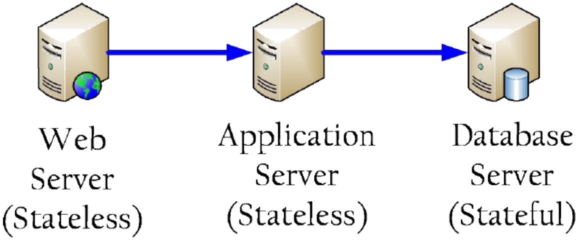


The Cloud is great for hosting Web applications

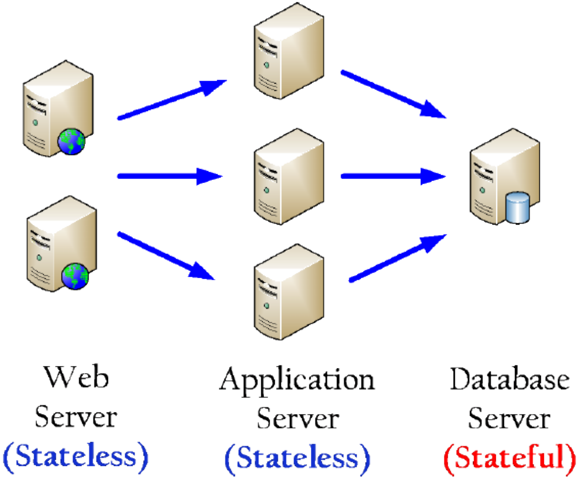


- “Infinite” number of computing resources
- Pay-as-you-go
- Resource provisioning

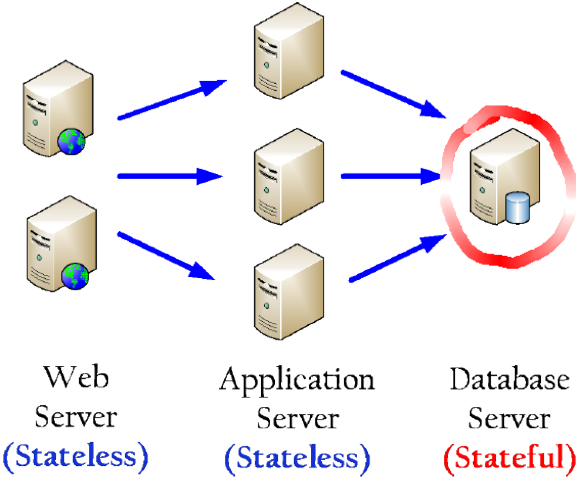
Web applications



Web applications



Web applications



Scaling relational databases

- Relational databases have many benefits:
  - A very powerful query language (SQL)
  - Strong consistency
  - Mature implementations
  - Well-understood by developers
  - Etc.

Scaling relational databases

- Relational databases have many benefits:
  - A very powerful query language (SQL)
  - Strong consistency
  - Mature implementations
  - Well-understood by developers
  - Etc.
- But also a few drawbacks:
  - Poor elasticity (ability to change the processing capacity easily)
  - Poor scalability (ability to process arbitrary levels of load)
  - Behavior in the presence of network partitions

Elasticity of relational databases

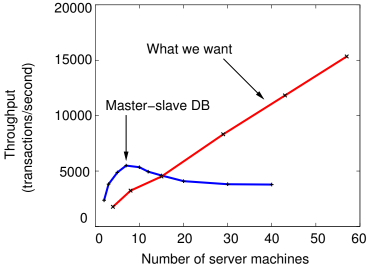
- Relational databases were designed in the 1970s
  - Designed for mainframes (a single super-expensive machine)
  - Not for clouds (many weak machines being created/stopped at any time)
- Master-slave replication:
  - 1 master database processes and serializes all updates
  - N slaves receive updates from the master and process all reads
  - Designed mostly for fault-tolerance, not performance
- How can we add a replica at runtime?
  - Take a snapshot of the database (very well supported by relational databases)
  - Copy the snapshot into the new replica
  - Apply all updates received since the snapshot
  - Add the new replica in the load balancing group

Elasticity of relational databases

- Relational databases were designed in the 1970s
  - Designed for mainframes (a single super-expensive machine)
  - Not for clouds (many weak machines being created/stopped at any time)
- Master-slave replication:
  - 1 master database processes and serializes all updates
  - N slaves receive updates from the master and process all reads
  - Designed mostly for fault-tolerance, not performance
- How can we add a replica at runtime?
  - Take a snapshot of the database (very well supported by relational databases)
  - Copy the snapshot into the new replica
  - Apply all updates received since the snapshot
  - Add the new replica in the load balancing group
  - This may take **hours** depending on the size of the database

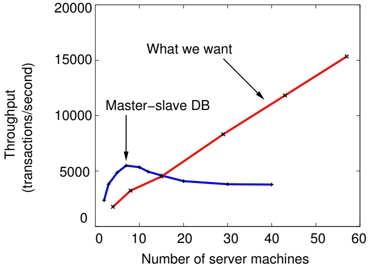
Scalability of relational databases

- Assuming an unlimited number of machines, can we process arbitrary levels of load?



Scalability of relational databases

- Assuming an unlimited number of machines, can we process arbitrary levels of load?



- Problem: full replication
  - Each replica must process every update
- Solution: partial replication
  - Each server contains a fraction of the total data
  - Updates can be confined to a small number of machines

Sharding

- Sharding = shared nothing architecture
- The programmer splits the database into independent partitions
  - Customers A-M → Database server 1
  - Customers N-Z → Database server 2
- Advantage: scalability
  - Each partition can work independently without processing the updates of other partitions
- Drawback: all the work is left for the developer
  - Defining the partition criterion
  - Routing requests to the correct servers
  - Implementing queries which span multiple partitions
  - Implementing elasticity
  - Etc.

Implementing sharding correctly is very difficult!

Hash Tables

- A Distributed Hash Table is a special kind of Hash Table
  - A hash table stores a large number of `(key,value)` pairs
  - Two very efficient operations:
    - `PUT(key, value)`
    - `value = GET(key)`
  - All other operations are unsupported (or extremely inefficient)
    - E.g., find all keys whose value contains "hello"

- A hash table is normally stored in a single computer
  - The storage is divided into N buckets
  - A (key,value) pair is stored in bucket $b = hash(key)\,\%\,N$

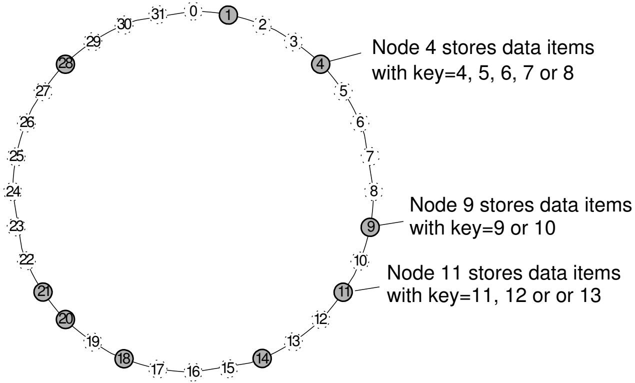
- A Distributed Hash Table uses multiple computers to store its content
  - Each computer stores only 1 bucket

Distributed Hash Tables

- See set of slides on Pastry

The Chord DHT

- The Chord DHT is organized as a logical ring
  - Each node is assigned a random m-bit identifier
  - Each data item is assigned a unique m-bit key
  - Entity with key k falls under jurisdiction of node with smallest $id \geq k$ (called its successor).



Why is this ring structure interesting?

- Automatic data partitioning
- Automatic load balancing
- Adding a new node does not disrupt the whole system
  - We just need to split one zone

Finding which node is in charge of which key

- Bad solution #1: let each node know the full list of other nodes
  - Each time a node joins or leaves we must replicate this information
  - Nasty consistency problem...

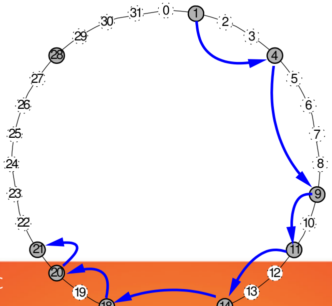
Finding which node is in charge of which key

- Bad solution #1: let each node know the full list of other nodes
  - Each time a node joins or leaves we must replicate this information
  - Nasty consistency problem...
- Bad solution #2: Let each node know only its own successor

☺ Local update when adding/removing nodes

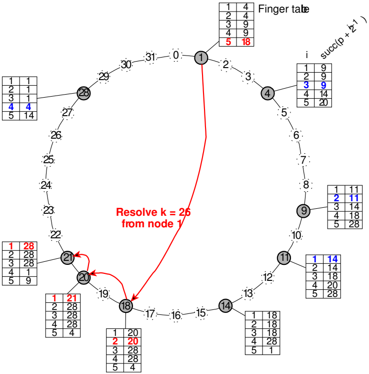
☹ But finding data is very expensive

Node 1 looks for item 24



Routing queries in Chord

- Chord nodes maintain more links than just their successor
  - 1/2 ring away, 1/4 ring away, 1/8 ring away, etc.
- Good properties:
  - Each node maintains $log_2(N)$ links (i.e., easy maintenance)
  - Each query is routed in $log_2(N)$ hops (i.e., efficient routing)



The two meanings of "Consistency"

1. For database experts: Consistency == Referential integrity in a single database
   - To make things simple: unique keys are really unique, foreign keys map on something etc.
   - This is the "C" from ACID

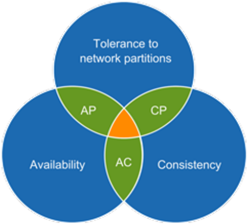
2. For distributed systems experts: Consistency = a property of replicated data
   - To make things simple: all copies of the same data seem to have the same value at any time

The CAP Theorem

- In a distributed system we want three important properties:
  1. Consistency: readers always see the result of previous updates
  2. Availability: the system always answers client requests
  3. Partition tolerance: the system doesn't break down if the network gets partitioned

The CAP Theorem

- In a distributed system we want three important properties:
  1. Consistency: readers always see the result of previous updates
  2. Availability: the system always answers client requests
  3. Partition tolerance: the system doesn't break down if the network gets partitioned
- Brewer's theorem: you cannot get all three at the same time
  - You must pick at most two out of three



- Relational databases usually implement AC

NoSQL takes the problem upside down

- NoSQL is designed with scalability in mind:
  - The database must be elastic
  - The database must be fully scalable
  - The database must tolerate machine failures
  - The database must tolerate network partitions

NoSQL takes the problem upside down

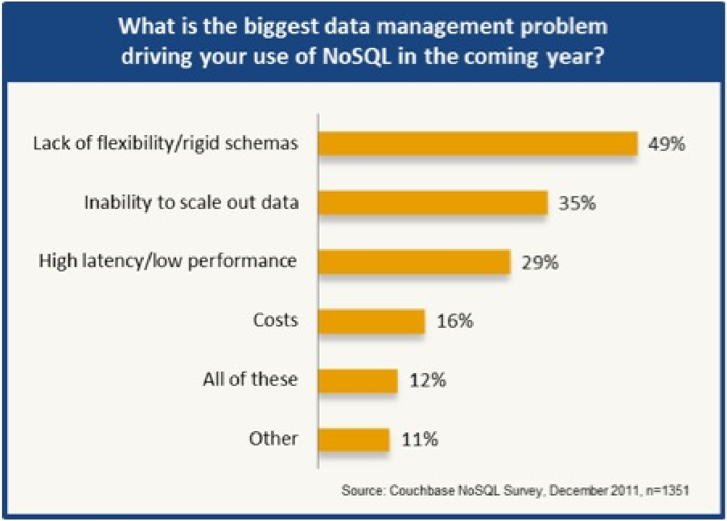
- NoSQL is designed with scalability in mind:
  - The database must be elastic
  - The database must be fully scalable
  - The database must tolerate machine failures
  - The database must tolerate network partitions
- What's the catch?
  - NoSQL must choose between AP and CP
    - Most NoSQL systems choose AP: they do not guarantee strong consistency
  - NoSQL do not support complicated queries
    - They do not support the SQL language
    - Only very simple operations!
- Different NoSQL systems apply these principles differently

NoSQL data stores rely on DHT techniques

- NoSQL data stores split data across nodes...
  - Excellent elasticity and scalability
- ... and replicate each data item on m nodes
  - For fault-tolerance
- If the network gets partitioned: serve requests within each partition
  - The system remains available
  - But clients will miss updates issued in the other partitions (bad consistency)
  - When the partition is resolved, updates from different partitions get merged

Flexible consistency models

- Some NoSQL data stores allow users to define the level of consistency they want
  - Replicate each data item over N servers
  - Associate each data item with a timestamp
  - Issue writes on all servers, consider a write to be successful when m servers have acknowledged
  - Read data from at least n servers (and return the freshest version to the client)
- If $m + n > N$ then we have strong consistency
  Quorum System
  - For example: $m = N$, $n = 1$
  - But other possibilities exist: $m = 1$, $n = N$
  - Or anything in between: $m = \frac{N}{2} + 1$, $n = \frac{N}{2} + 1$
- If $m + n \leq N$ then we have weak consistency
  - Faster


What is the biggest data management problem driving your use of NoSQL in the coming year?
Lack of flexibility/rigid schemas
49%
Inability to scale out data
35%
High latency/low performance
29%
Costs
16%
All of these
12%
Other
11%
Source: Couchbase NoSQL Survey, December 2011, n=1351

Flexible data schemas

- In NoSQL data stores there is no need to impose a strict data schema
  - Anyway the data store treats each row as a `(key,value)` pair
  - No requirement for the `value` $\Rightarrow$ no fixed data schema
  - Not the same as empty values!

```
{
   FirstName:"Bob",
   Address:"5 Oak St.",
   Hobby:"sailing"
}
```

```
{
    FirstName:"Jonathan",
    Address:"15 Wanamassa Point Road",
    Children:[
       {Name:"Michael",Age:10},
       {Name:"Jennifer", Age:8},
       {Name:"Samantha", Age:5},
       {Name:"Elena", Age:2}
  ]
 }
```

Example: AppEngine's Datastore

AppEngine's Datastore relies on Google BigTable
(the first NoSQL database: OSDI 2006)

- You can only GET and PUT entities based on their key
  - No complex query
- Entities are organized into entity groups
  - Operations within one entity group are strongly consistent
  - Operations spanning multiple entity groups are weakly consistent
- The datastore supports at most 1 update per second per entity group
  - Entity groups are replicated using Paxos across multiple machines in different data centers
  - ☺ Guaranteed strong consistency even if nodes misbehave in strange ways
  - ☹ Paxos is known to be very slow

Example: Dynamo

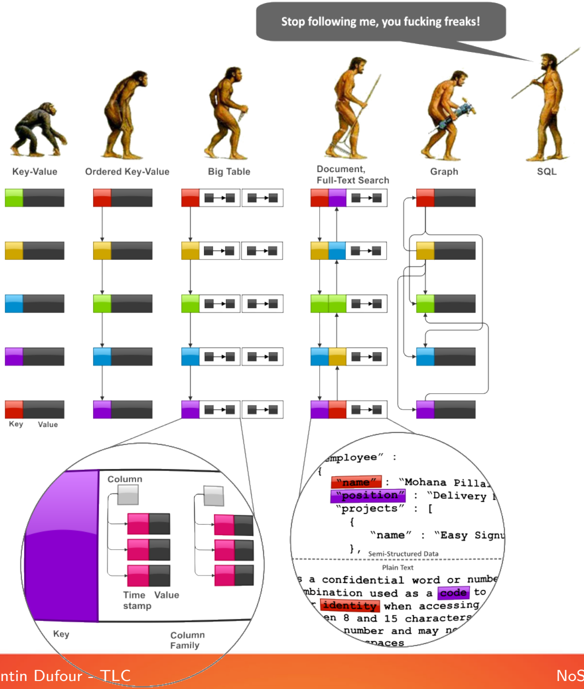
- See set of slides on Dynamo.

Data modeling for NoSQL datastores

- Data normalization techniques will not work for NoSQL
  - Forget UML and other related methodologies
- There is very little formal work on data schema design for NoSQL :-(
  - NoSQL is too young for that
  - Each NoSQL datastore has specific features
- But there exists useful guidelines
  - Keeping in mind that each NoSQL datastore has specific functionality
  - Exploit them to the fullest extent!

Different types of NoSQL datastores

- Key-value stores do not attempt to interpret the content of values
  - PUT(key,value)
  - value=GET(key)
  - DELETE(key)
  - Examples: AppEngine's datastore, HBase, AWS Dynamo
- Ordered key-value stores let you iterate through keys
  - Examples: Scalarix
- Document databases do interpret the content of values
  - Impose a syntax for values (JSON, XML, etc.)
  - Support value-based operations (e.g., secondary-key queries)
    - With various performance behaviors depending on the database
  - Example: CouchDB, Apache Cassandra
- More exotic types of data stores: graph databases, object databases, etc.


Stop following me, you fucking freaks!
Key-Value
Ordered Key-Value
Big Table
Document,
Full-Text Search
Graph
SQL
Key
Value
Column
Time stamp
Value
Key
Column Family
Semi-Structured Data
Plain Text

Common properties

Let's compare Amazon's SimpleDB,
Google's BigTable and Yahoo's PNUTS

- Data are organized in tables
- A table contains a number of data items identified by a primary key
- Data items are organized as a collection of key-value pairs
  - Only data type: string
  - Data items from the same table do not necessarily have the same list of attributes (flexible data schema)
- Data items are accessed by PUT/GET using their primary key
- No supported operation across tables (such as joins)

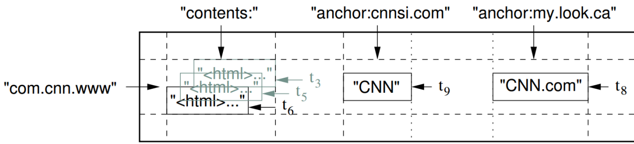
Amazon's SimpleDB / Apache's Cassandra

- SimpleDB allows records to contain multiple values with the same key (e.g., a multiset)
- Data are organized into "domains"
  - Domains $\sim$ tables
  - No schema
- SimpleDB supports range queries
- Consistency: eventual consistency
  - Also some form of strong consistency is supported (with lower levels of performance)

Google's BigTable / Apache's HBase

- Columns are organized in column families: `"family:column_name"`
  - Column families are the granularity for access control

- Tables have more dimensions than the standard model
  - Values are indexed by row, column and timestamp
  - (row:string, column:string, time:int64) $\rightarrow$ string



- Rows are sorted
  - BigTable allows users to iterate through records
  - ... or through successive versions of the same record

Yahoo's PNUTS

- PNUTS requires an explicit list of attributes per record (i.e., a schema)
  - But it is not necessary to use all attributes
  - And it is easy to change the list at runtime
- UPDATE, DELETE and INSERT queries must specify a primary key
- Tables can be hashed or ordered
  - Hashed: excellent load balancing, efficient primary-key queries
  - Ordered: less good load balancing, but support for range queries
  - In both cases: PNUTS supports "multiget" queries to retrieve several records in parallel (from one or more tables)
- Consistency: single-row transactions

Comparison

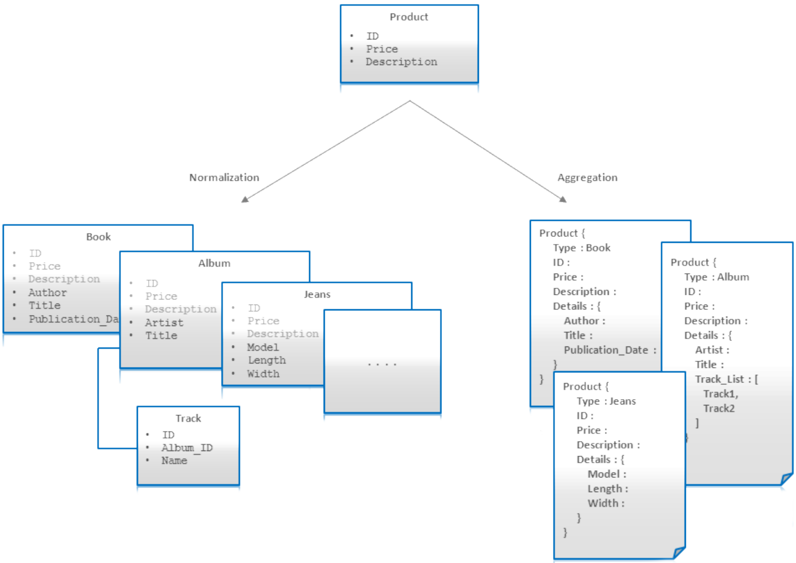
	Amazon's SimpleDB	Google's Bigtable	Yahoo's PNUTS
Data Item	Multi-value attribute	Multi-version with timestamp	Multi-version with timestamp
Schema	No schema	Column-families	Explicitly claimed attributes
Operation	Range queries on arbitrary attributes of a table	Single-table scan with various filtering conditions	Single-table scan with predicates
Consistency	Eventual consistency	Single-row transaction	Single-row transaction

Denormalization

- Normalization defines data stuctures regardless of the queries
  - Hidden assumption: if the data are well-organized we can always query them easily
  - This is true for SQL databases but not for NoSQL datastores
- Denormalization does the opposite of normalization: structure data according to future queries
  - Group all data necessary for a query at the same place
  - We often end up copying the same data at multiple places in the datastore
  - ☺ Excellent performance if we do things well
  - ☹ Database consistency issues: all updates must be applied everywhere, it is easy to introduce mistakes

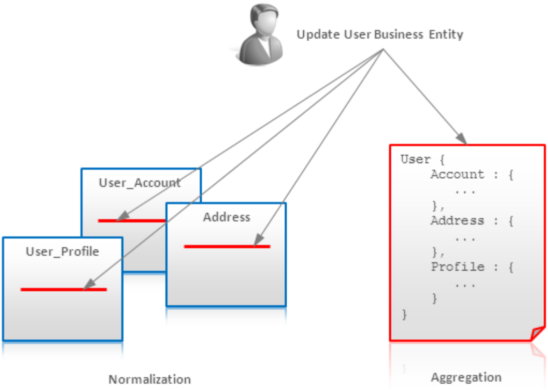
Aggregates

- NoSQL datastores allow flexible data schemas
  - Stored values may have complex nested structures
  - No need to pre-define these structures, we can simply create them at runtime
  - Each record may have a different structure
- Example 1: a `User` record links to the list of his `Messages`
  - Normalized version: two tables (`Users` and `Messages`) with references between the two
  - NoSQL version: insert the entire messages inside the User record
- Example 2: different types of products
  - Normalized verson: one table for each type of product (with its specific structure)
  - NoSQL version: store all products with their specificities next to each other


Product
ID
Price
Description
Normalization
Aggregation
Book
ID
Price
Description
Author
Title
Publication_Da
Album
ID
Price
Description
Artist
Title
Jeans
ID
Price
Description
Model
Length
Width
. . . .
Track
ID
Album_ID
Name
Product {
Type : Book
ID :
Price :
Description :
Details : {
Author :
Title :
Publication_Date :
}
}
Product {
Type : Album
ID :
Price :
Description :
Details : {
Artist :
Title :
Track_List : [
Track1,
Track2
]
Product {
Type : Jeans
ID :
Price :
Description :
Details : {
Model :
Length :
Width :
}
}

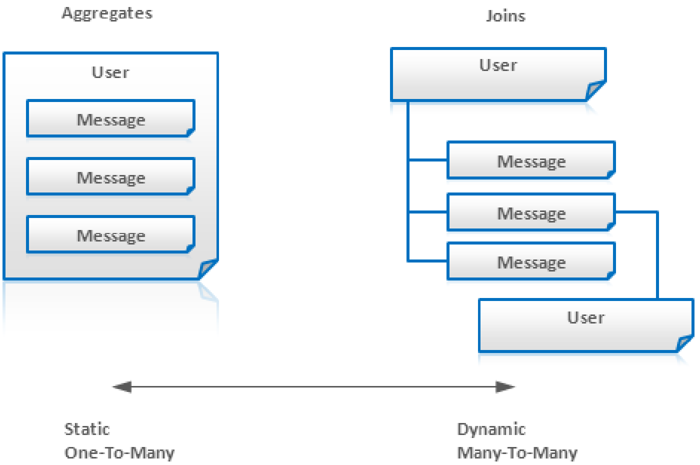
Atomic aggregates

- Aggregates have one nice side-effect: atomic updates
  - NoSQL datastores often support atomic updates per data item
  - But they rarely support multi-item transactions
- If multiple updates are located in the same record they become atomic



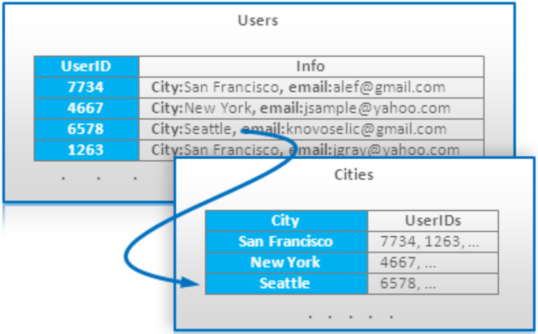
Application-side joins

- Very few NoSQL data stores support joins
  - Denormalization and aggregates often allow us to avoid joins
- But sometimes we cannot avoid joins
  - Many-to-many relationships between records
  - Frequently updated data items
- Solution: application-side joins
  - Let the application fetch all necessary data items
  - Join them by hand


Aggregates
Joins
User
Message
Message
Message
User
Message
Message
Message
User
Static
One-To-Many
Dynamic
Many-To-Many

Index tables

- We can implement foreign keys by simply building index tables
  - Replace one join query with 2 simple queries
  - Beware: you lose atomicity

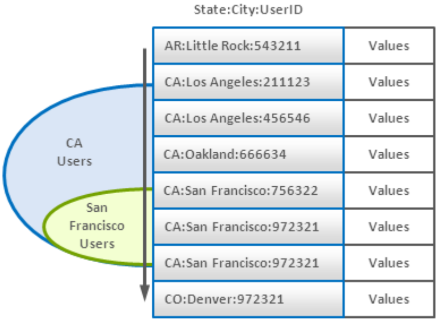


Enumerable keys

- DHTs normally hash keys before deciding where to store each data item
  - Excellent for load balancing
  - But contiguous keys end up being located in random nodes in the system
- Some NoSQL decided to drop hashing
  - Much less efficient for load balancing
  - But it allows applications to iterate through keys
- You can embed information in the keys
  - Example: key=*userID_messageID*
  - You can easily access all messages from a user: start at UserID_0 and iterate

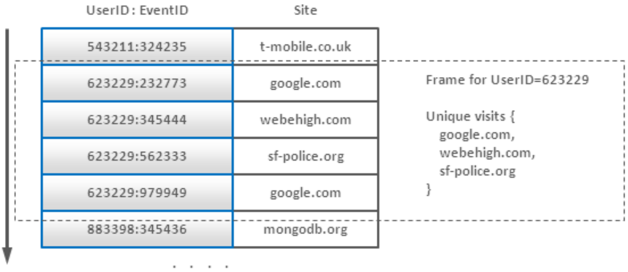
Composite key index

- We can combine index tables with fancy key structures
  - This often allows for efficient secondary-key queries
- Example: select users by their location
  - `SELECT * FROM users WHERE state="CA"`
  - `SELECT * FROM users WHERE city="San Francisco"`
  - NoSQL solution: design keys as State:City:UserID



Aggregation with Composite Keys

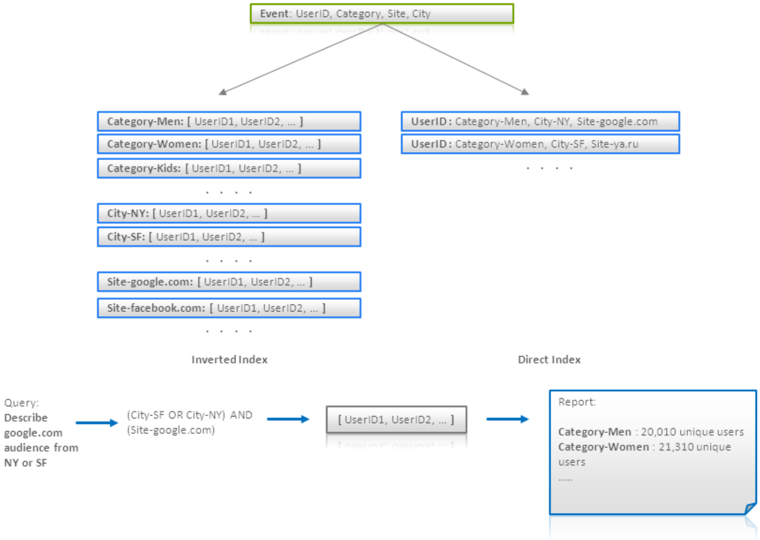
- We can also use composite keys for data aggregation
- Example: search a log file for all unique sites visited by a user
  - `SELECT count(distinct(user_id)) FROM clicks GROUP BY site`
  - NoSQL solution: make sure to keep contiguous log records per user
    - And then eliminate redundancy in the application itself



- This is much more efficient than keeping log entries from each user in a single record

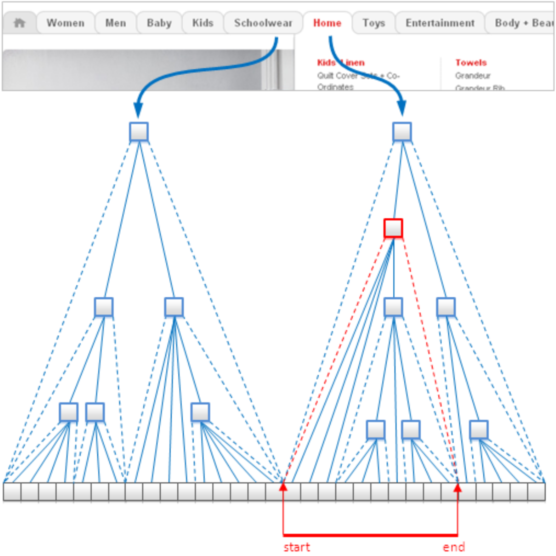
Inverted search

- If we want to search items along multiple criteria we cannot use composite keys
  - With composite keys we can support only one type of search
- Example: we want to search users by their gender, city, the sites they visit etc.
  - NoSQL solution: build inverted indexes explicitly
  - Key=property; Value=reference to the main table


Event: UserID, Category, Site, City
Category-Men: [UserID1, UserID2, ...]
Category-Women: [UserID1, UserID2, ...]
Category-Kids: [UserID1, UserID2, ...]
. . . .
City-NY: [UserID1, UserID2, ...]
City-SF: [UserID1, UserID2, ...]
. . . .
Site-google.com: [UserID1, UserID2, ...]
Site-facebook.com: [UserID1, UserID2, ...]
. . . .
UserID : Category-Men, City-NY, Site-google.com
UserID : Category-Women, City-SF, Site-ya.ru
. . . .
Inverted Index
Direct Index
Query:
Describe google.com audience from NY or SF
(City-SF OR City-NY) AND (Site-google.com)
[UserID1, UserID2, ...]
Report:
Category-Men : 20,010 unique users
Category-Women : 21,310 unique users
.....

Nested sets

- How do we represent a hierarchical structure in NoSQL?
  - Bad solution #1: store the entire tree in one data item
  - Bad solution #2: store each node separately, maintain a list of children in all non-leaf nodes
- Solution: nested sets
  - Map each leaf to one data item in the NoSQL store
  - Make each non-leaf node maintain the beginning/end index
    - Very efficient for read/search
    - Not so efficient for updates


Women
Men
Baby
Kids
Schoolwear
Home
Toys
Entertainment
Towels
Grandeur
start
end

Using MapReduce for complex queries

- Some queries can be unfrequent but very complex
  - E.g., data mining queries
- You cannot redesign your entire data schema for just one ad-hoc query
- Implementing the entire query in the application can be inefficient
  - In the worst case: fetch the entire data store on the client, let the client process the query locally
- Solution: MapReduce
  - Example: MongoDB is fully integrated with MapReduce
  - You can request a MapReduce job over the content of the datastore in just one command

MapReduce queries in MongoDB

```
db.runCommand(
 { mapreduce : <collection>,
   map : <mapfunction>,
   reduce : <reducefunction>,
   out : <see output options below>
   [, query : <query filter object>]
   [, sort : <sorts the input objects using this key. Useful for optimization, like sorting by
the emit key for fewer reduces>]
   [, limit : <number of objects to return from collection, not supported with sharding>]
   [, keeptemp: <true|false>]
   [, finalize : <finalizefunction>]
   [, scope : <object where fields go into javascript global scope >]
   [, jsMode : true]
   [, verbose : true]
 }
);
```

Example [1/2]

```
$ ./mongo
> db.things.insert( { _id : 1, tags : ['dog', 'cat'] } );
> db.things.insert( { _id : 2, tags : ['cat'] } );
> db.things.insert( { _id : 3, tags : ['mouse', 'cat', 'dog'] } );
> db.things.insert( { _id : 4, tags : []  } );

> // map function
> m = function(){
...     this.tags.forEach(
...         function(z){
...             emit( z , { count : 1 } );
...         }
...     );
...};

> // reduce function
> r = function( key , values ){
...     var total = 0;
...     for ( var i=0; i<values.length; i++ )
...         total += values[i].count;
...     return { count : total };
...};
```

Example [2/2]

```
> res = db.things.mapReduce(m, r, { out : "myoutput" } );
> res
{
        "result" : "myoutput",
        "timeMillis" : 12,
        "counts" : {
                "input" : 4,
                "emit" : 6,
                "output" : 3
        },
        "ok" : 1,
}
> db.myoutput.find()
{"_id" : "cat" , "value" : {"count" : 3}}
{"_id" : "dog" , "value" : {"count" : 2}}
{"_id" : "mouse" , "value" : {"count" : 1}}

> db.myoutput.drop()
```

Conclusion

- NoSQL datastores are designed for scalability
  - Even at the cost of reducing the set of offered functionalities
- Different NoSQL data stores can have very different properties
  - It is important to understand these specific functionalities to make the best use of each system
  - Also useful for choosing one datastore (when possible)
- Very little theoretical background on how to organize data
  - But there exists useful guidelines