

# CouchDB and Lucene Index

Document processing for data mining

Presented by Sijing Yang

# CouchDB

A NoSQL DBMS that does not mimic SQL

# Overview

---

- Background
  - SQL vs. NoSQL
  - A refresh on MongoDB
- Introduction
  - What is CouchDB and Why
- Data modeling
  - CouchDB vs. MongoDB
- Query capabilities
  - CouchDB vs. MongoDB
- Concurrency control and Distributed architecture
- Conclusions and Some facts

# Background: SQL vs. NoSQL

---

- **Data modeling: schema-less**
  - Relational: What answers do we have? (driven by the structure of available data)
  - NoSQL: What questions do we have? (driven by application-specific pattern)
- **Query capability:**
  - Relational: human user-oriented, query is simple
  - NoSQL: application-oriented, query is comparatively complex
- **Scalability:**
  - Relational: vertical
  - NoSQL: horizontal

# Background: SQL vs. NoSQL

---

- NoSQL is naturally fit for big data.
  - Unstructured data with similar semantics but varied syntax
  - Large volume of data for which scalability is becoming a must and consistency expensive

# Visual Guide to NoSQL Systems

**Availability:**  
Each client can  
always read  
and write.

**A**

**Data Models**

Relational (comparison)  
Key-Value  
Column-Oriented/Tabular  
Document-Oriented

**CA**

RDBMSs  
(MySQL,  
Postgres,  
etc)

Aster Data  
Greenplum  
Vertica

**AP**

Dynamo  
Voldemort  
Tokyo Cabinet  
KAI

Cassandra  
SimpleDB  
CouchDB  
Riak

**Pick Two**

**C**

**Consistency:**  
All clients always  
have the same view  
of the data.

**CP**

BigTable  
Hypertable  
Hbase

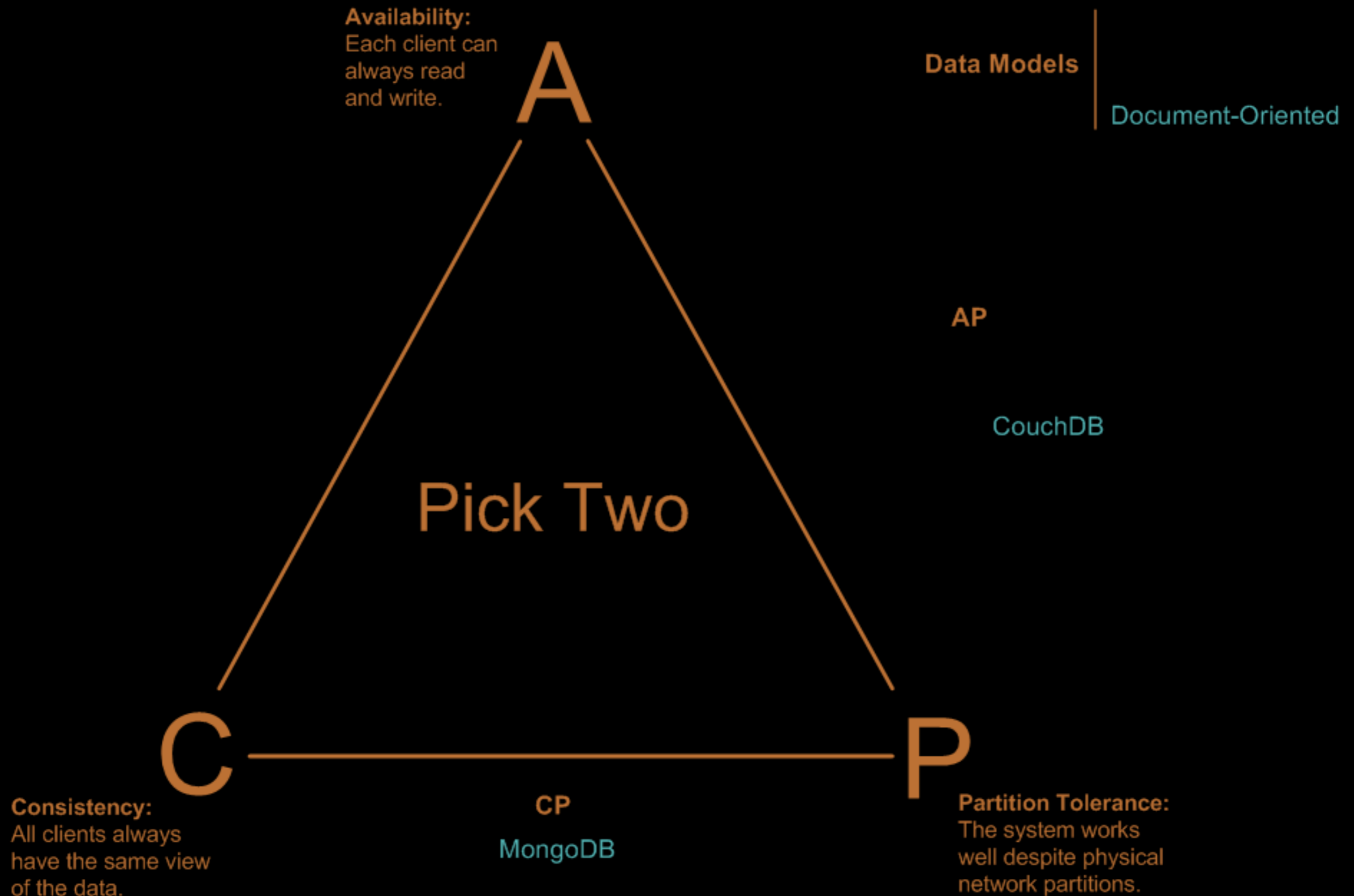
MongoDB  
Terrastore  
Scalaris

Berkeley DB  
MemcacheDB  
Redis

**P**

**Partition Tolerance:**  
The system works  
well despite physical  
network partitions.

# Visual Guide to NoSQL Systems

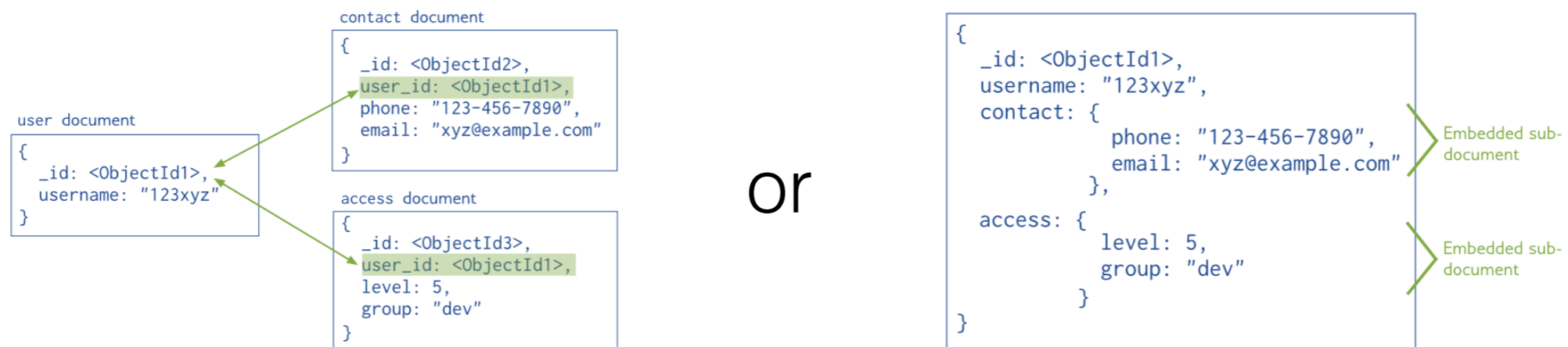


# Background: A refresh on MongoDB

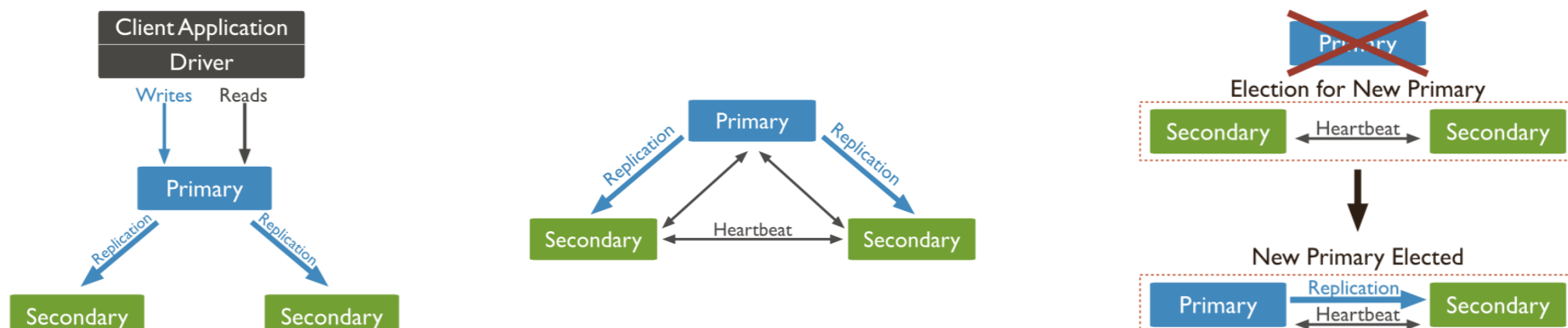
- Use collections to organize modules



- Normalized (Reference) or denormalized (embedding)



- Strict consistency (All writes must go to primary node)





# Introduction: What is CouchDB?

---



- Name comes from:
  - Cluster Of Unreliable Commodity Hardware
  - Relax (in a couch)
- Written in Erlang, initial release in 2005
- Licence: Apache, Original author: Damien Katz, et al.

# Introduction: What is CouchDB?

---

- An open source, document-oriented, NoSQL database that uses JSON to store data, JavaScript as its query language, and HTTP for an API.
- Instead of locking mechanism, CouchDB uses MVCC to resolve conflicts, and incremental replication to achieve eventual consistency.

# Introduction: Why CouchDB?

---

- Availability, Locality and Scalability

Each node in a system should be able to make decisions purely based on local state. If you need to do something under high load with failures occurring and you need to reach agreement, you're lost. If you're concerned about scalability, any algorithm that forces you to run agreement will eventually become your bottleneck. Take that as a given.

—Werner Vogels, Amazon CTO and Vice President

- “A database that completely embraces the web.”

# Data Modeling of CouchDB: Overview

---

- JSON format
- Self-contained data (as opposed to referenced data)
- One big store of documents, no collection layer
- B tree storage engine + MapReduce to compute results of a view

# Data Modeling of CouchDB: JSON Format

---

- CouchDB: JSON

- MongoDB: BSON

BSON is binary JSON



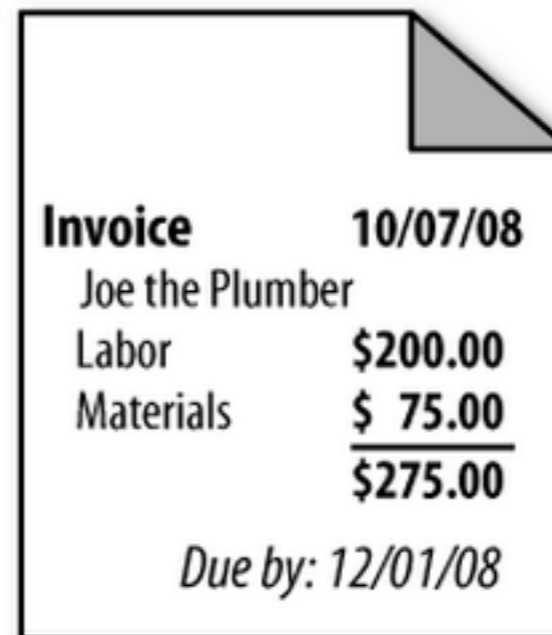
BSON is a JSON that has been serialized as a binary document.

# Data Modeling of CouchDB: Self-contained Data

---

- CouchDB: purely self-contained (*Say Goodbye to SQL*)
- MongoDB: embedded (*NoSQL*); or referenced (*SQL-like*)

Real-world data is managed as real-world documents

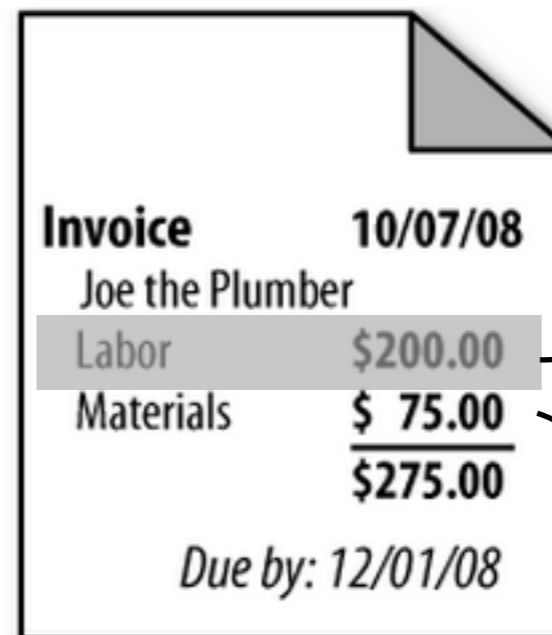


<b>Invoice</b>	<b>10/07/08</b>
Joe the Plumber	
Labor	<b>\$200.00</b>
Materials	<b>\$ 75.00</b>
	<b>\$275.00</b>
<i>Due by: 12/01/08</i>	

# Data Modeling of CouchDB: Self-contained Data

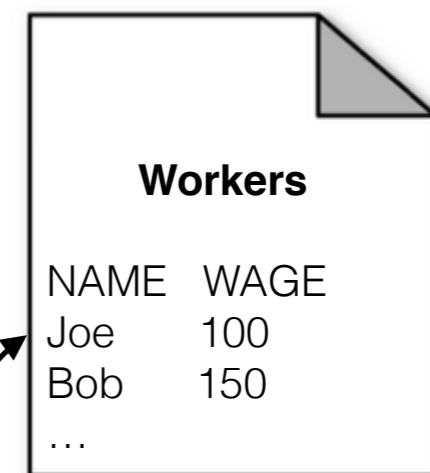
- CouchDB: purely self-contained (*Say Goodbye to SQL*)
- MongoDB: embedded (*NoSQL*); or referenced (*SQL-like*)

If real-world data is not managed as real-world data



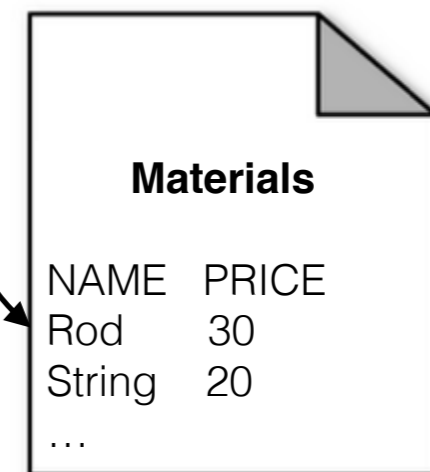
An invoice document for Joe the Plumber dated 10/07/08. It lists Labor at \$200.00 and Materials at \$75.00, with a total of \$275.00. The due date is 12/01/08. Arrows point from the Labor and Materials rows to the Workers and Materials tables respectively.

Invoice		10/07/08
Joe the Plumber		
Labor	\$200.00	
Materials	\$ 75.00	
	<u>\$275.00</u>	
Due by: 12/01/08		



A table titled "Workers" with columns NAME and WAGE. It lists Joe with a wage of 100 and Bob with a wage of 150.

NAME	WAGE
Joe	100
Bob	150
...	



A table titled "Materials" with columns NAME and PRICE. It lists Rod with a price of 30 and String with a price of 20.

NAME	PRICE
Rod	30
String	20
...	

# Data Modeling of CouchDB: Data Storage

- CouchDB:  
one big warehouse  
No global indexes predefined on DB level, create a view to report results instead

- MongoDB:  
separated by collections  
Can create index for any field of documents in a collection (identical to indexing in RDBMS)

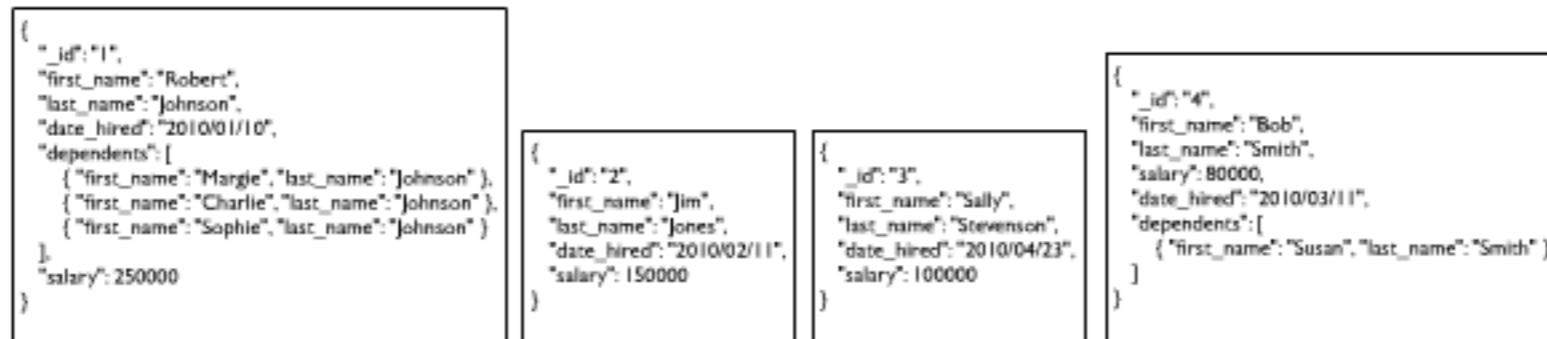




# Query Capabilities: How do you aggregate unstructured data?

---

- Define a view
  - Map takes documents and emits key/value pairs



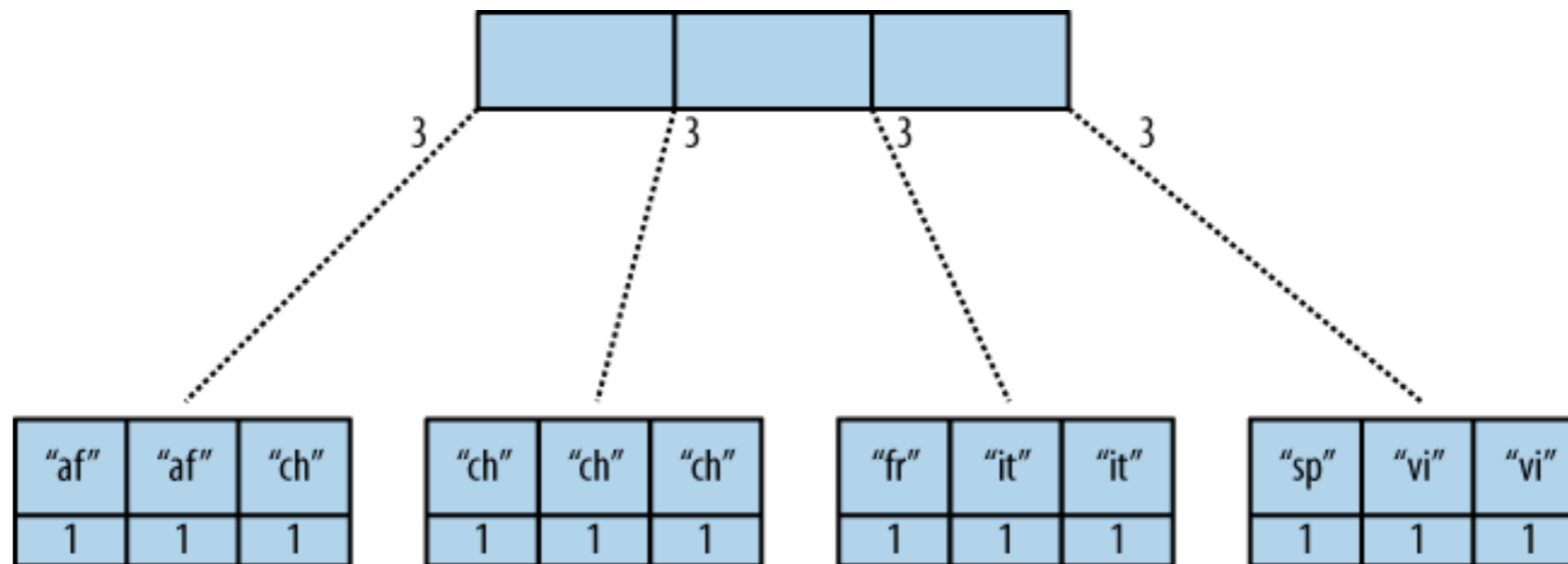
MapReduce

```
{"total_rows":4,"offset":0,"rows":[
{"id":"1","key":"1","value":{"first_name":"Margie","last_name":"Johnson"}},
{"id":"1","key":"1","value":{"first_name":"Charlie","last_name":"Johnson"}},
{"id":"1","key":"1","value":{"first_name":"Sophie","last_name":"Johnson"}},
{"id":"4","key":"4","value":{"first_name":"Susan","last_name":"Smith"}}
]}
```

# Query Capabilities: How do you aggregate unstructured data?

---

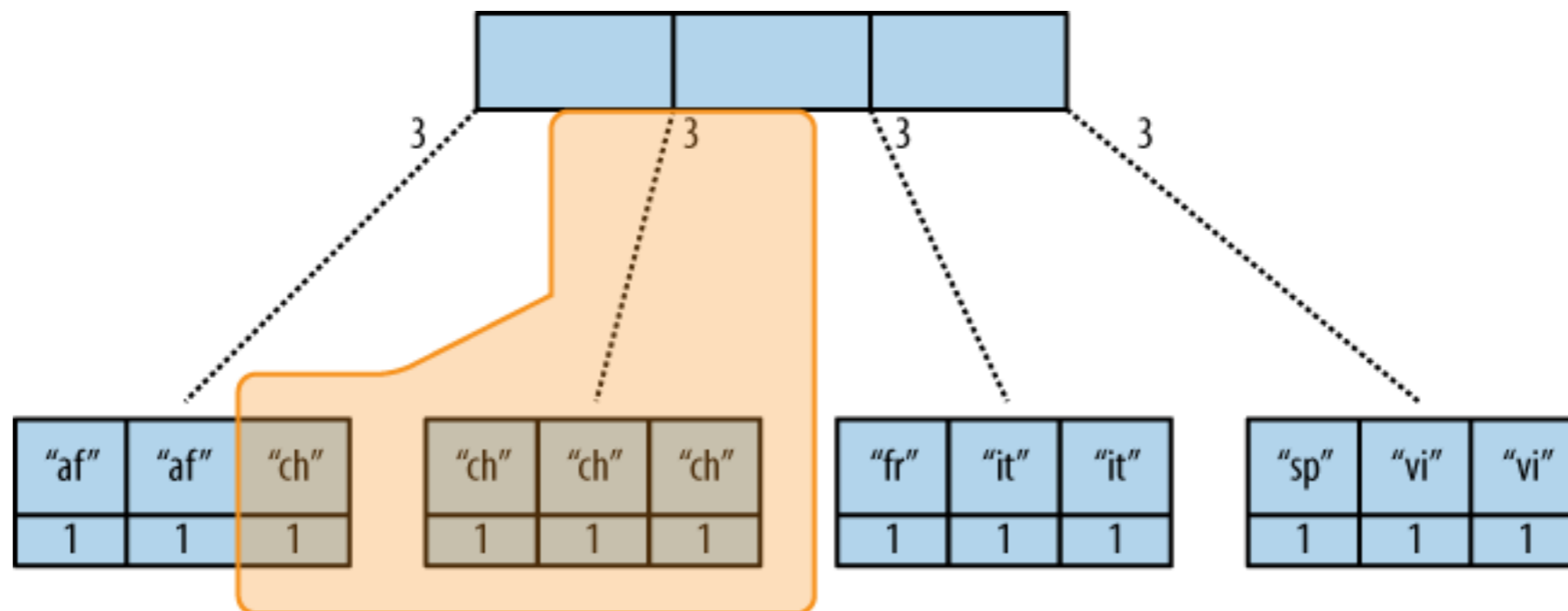
- Construct B-tree index
  - CouchDB storage engine constructs a B-tree index



# Query Capabilities: How do you aggregate unstructured data?

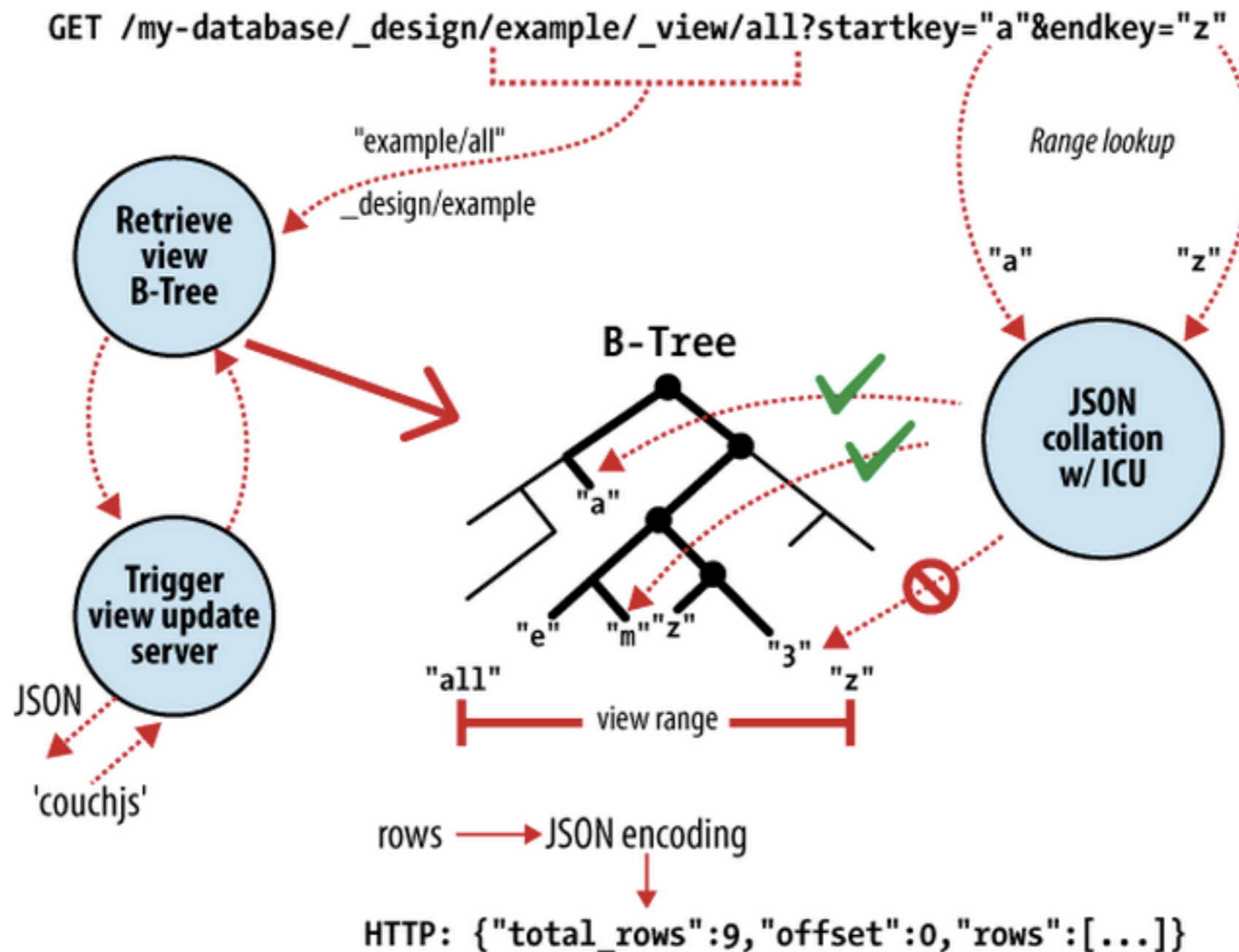
---

- Query the view
  - Reduce operates on the subtree to do aggregation



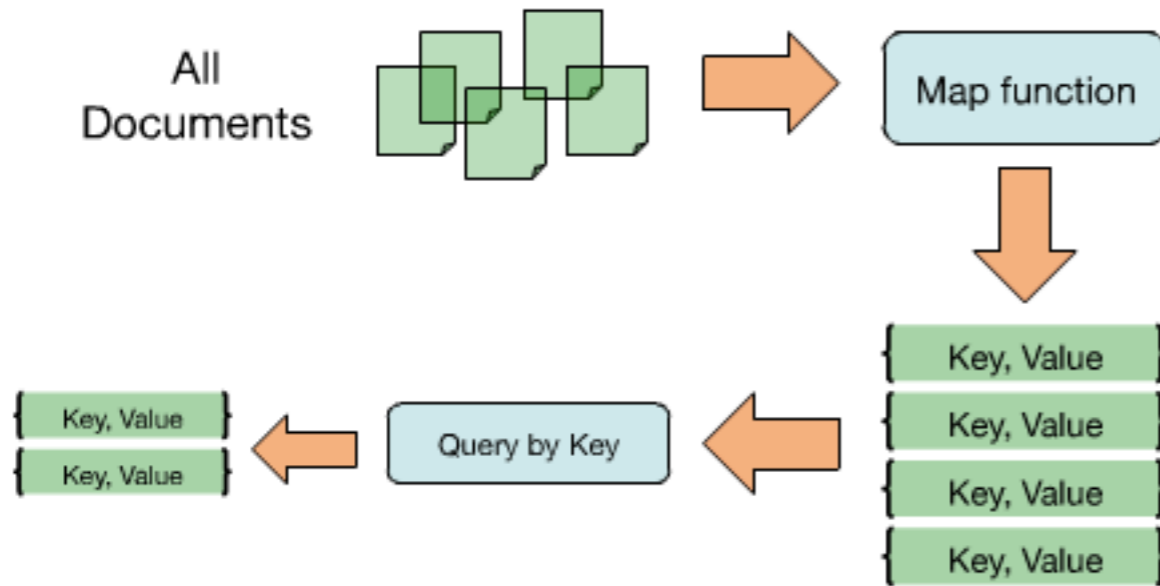
# Query Capabilities: How do you aggregate unstructured data?

- MapReduce + B-tree = results of a view



# Query Capabilities

- CouchDB:  
MapReduce(complex queries)

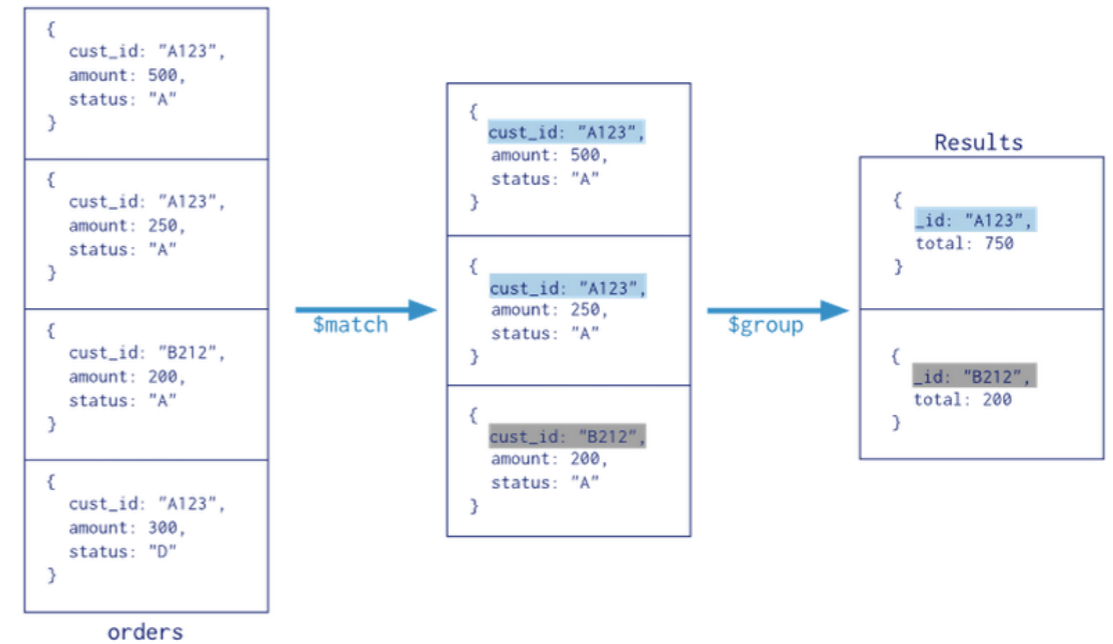


```

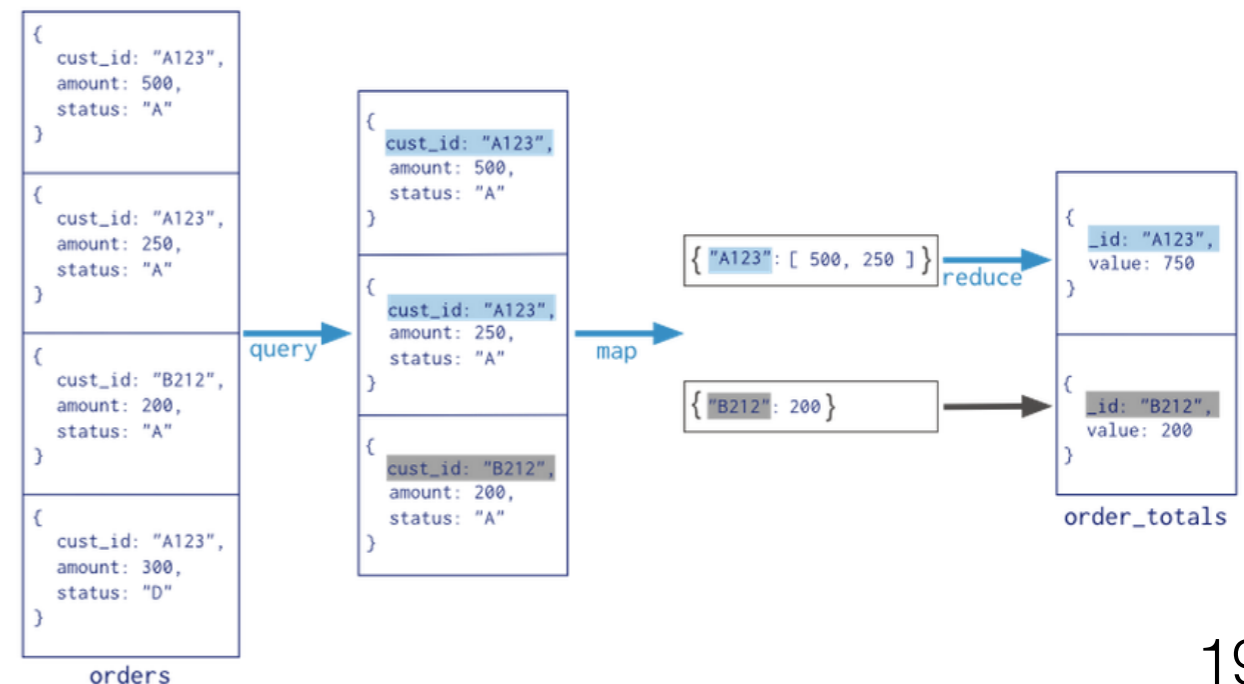
View Code
Map Function:
function(doc) {
  var store, price, value;
  if (doc.item && doc.prices){
    for (store in doc.prices){
      price = doc.prices[store];
      value = [doc.item, price];
      emit(price, value);
    }
  }
}
Run Language: javascript
  
```

Well, comparatively complex...

- MongoDB:  
(1) Aggregation pipeline(SQL-like)



- (2) MapReduce(complex queries)



# Data Management

- Futon: Built-in administration interface

The screenshot displays the Apache CouchDB Futon interface. The browser address bar shows the URL `127.0.0.1:5984/_utils/database.html?hello-world/_temp_view`. The interface includes a navigation bar with 'Overview' and 'hello-world'. Below this, there are buttons for 'New Document', 'Security...', 'Compact & Cleanup...', and 'Delete Database...'. A 'Jump to:' field is set to 'Document ID', and the 'View:' dropdown is set to 'Temporary view...'. A 'Stale views' checkbox is present.

The 'View Code' section shows a JavaScript map function:

```
function(doc) {
  var store, price, value;
  if (doc.item && doc.prices){
    for (store in doc.prices){
      price = doc.prices[store];
      value = [doc.item, price];
      emit(price, value);
    }
  }
}
```

Below the code editor are buttons for 'Run', 'Language: javascript', 'Revert', 'Save As...', and 'Save'. A warning message states: "Warning: Please note that temporary views are not suitable for use in production, as they are really slow for any database with more than a few dozen documents. You can use a temporary view to experiment with view functions, but switch to a permanent view before using them in an application."

The main content area displays a table with 'Key' and 'Value' columns. The table contains 9 rows of data:

Key	Value
0.79 ID: a688d8d20e17b5e87e47da6aa8002490	["apple", 0.79]
0.79 ID: a688d8d20e17b5e87e47da6aa800423d	["bananas", 0.79]
1.09 ID: a688d8d20e17b5e87e47da6aa8003415	["orange", 1.09]
1.59 ID: a688d8d20e17b5e87e47da6aa8002490	["apple", 1.59]
1.99 ID: a688d8d20e17b5e87e47da6aa8003415	["orange", 1.99]
1.99 ID: a688d8d20e17b5e87e47da6aa800423d	["bananas", 1.99]
3.19 ID: a688d8d20e17b5e87e47da6aa8003415	["orange", 3.19]
4.22 ID: a688d8d20e17b5e87e47da6aa800423d	["bananas", 4.22]
5.99 ID: a688d8d20e17b5e87e47da6aa8002490	["apple", 5.99]

At the bottom, it says 'Showing 1-9 of 9 rows' and 'Rows per page: 10'. The right sidebar contains the CouchDB logo, navigation links for 'Tools' (Overview, Configuration, Replicator, Status), 'Documentation' (Manual), 'Diagnostics' (Verify Installation), and 'Recent Databases' (hello-world).

# Data Management

---

- REST API: a thin wrapper around the DB core

## REST API

# Create

POST http://localhost:5984/employees

# Read

GET http://localhost:5984/employees/1

# Update

PUT http://localhost:5984/employees/1

# Delete

DELETE http://localhost:5984/employees/1

# Data Management

---

- REST API: a thin wrapper around the DB core

Welcome:

```
pocoyang: ~ $: curl http://127.0.0.1:5984/  
{ "couchdb": "Welcome", "uuid": "bd94a3f857e93302522f918d997cb706", "version": "1.6.1",  
  "vendor": { "version": "1.6.1-1", "name": "Homebrew" } }
```

Add a new database:

```
pocoyang: ~ $: curl -X PUT http://127.0.0.1:5984/albums  
{"ok": true}
```

Add a new document:

```
pocoyang: ~ $: curl -X PUT http://127.0.0.1:5984/albums/a688d8d20e17b5e87e47da6a  
a8004eaa -d '{"title": "D Minor K466", "artist": "Mozart"}'  
{"ok": true, "id": "a688d8d20e17b5e87e47da6aa8004eaa", "rev": "1-d067700c88a3a78e5863  
970ccad4f923"}
```

Get a new UUID:  
(if don't have one)

```
pocoyang: ~ $: curl -X GET http://127.0.0.1:5984/_uuids  
{"uuids": ["a688d8d20e17b5e87e47da6aa8004eaa"]}
```

Read a document:

```
pocoyang: ~ $: curl -X GET http://127.0.0.1:5984/albums/a688d8d20e17b5e87e47da6a  
a8004eaa  
{"_id": "a688d8d20e17b5e87e47da6aa8004eaa", "_rev": "1-d067700c88a3a78e5863970ccad4  
f923", "title": "D Minor K466", "artist": "Mozart"}
```

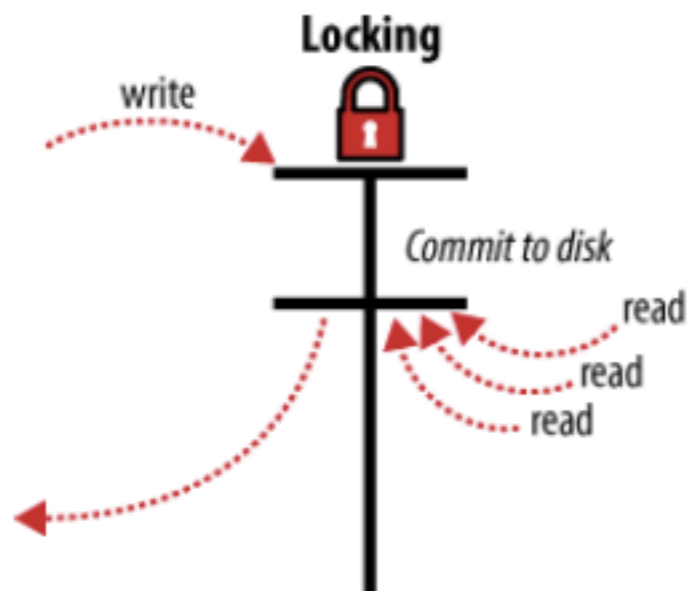
.....



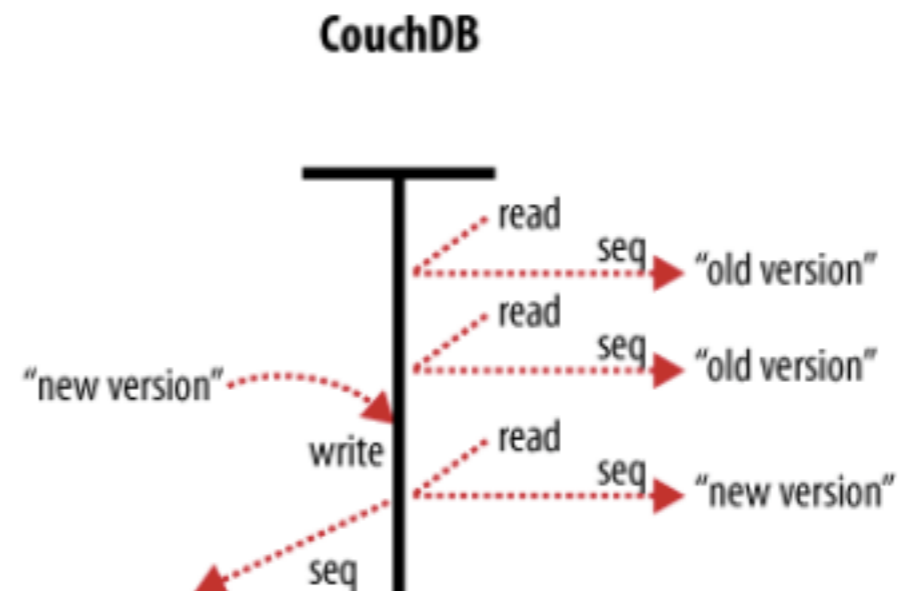
# Concurrency control of CouchDB

---

- Multi-Version Concurrency Control:
  - Doesn't rely on global state, always available to readers;
  - Each reader is reading the latest visible snapshot



- MongoDB

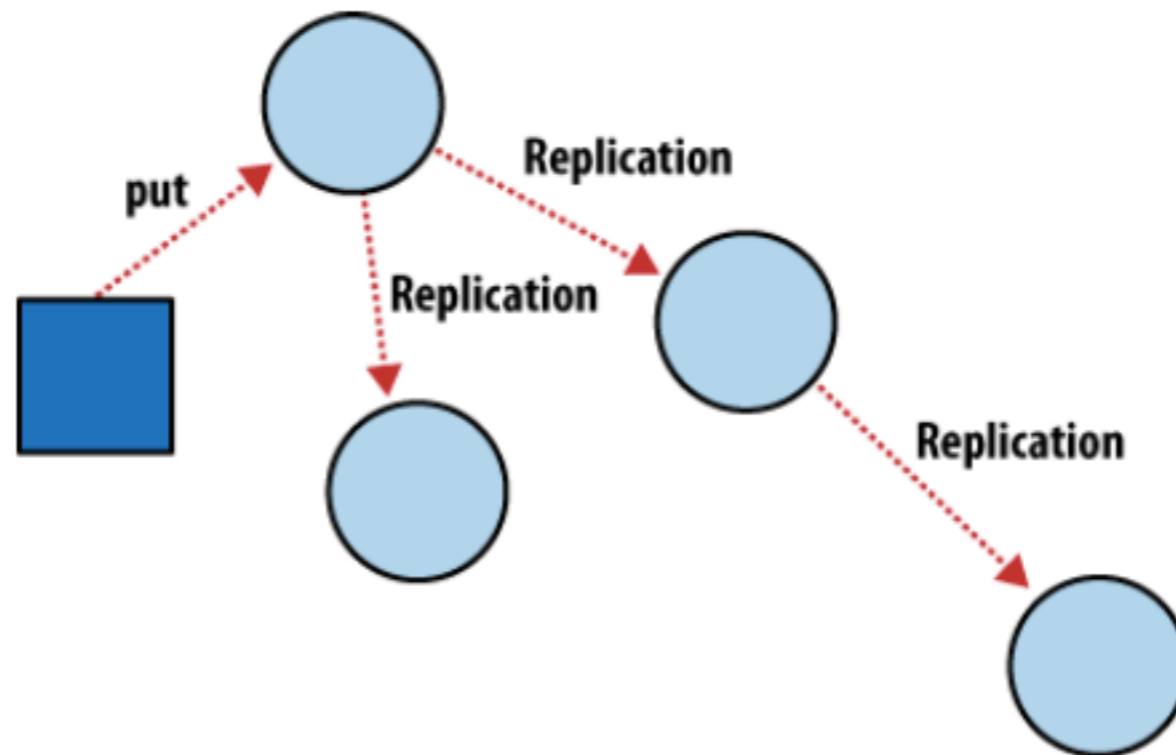


- CouchDB

# Distributed Architecture of CouchDB

---

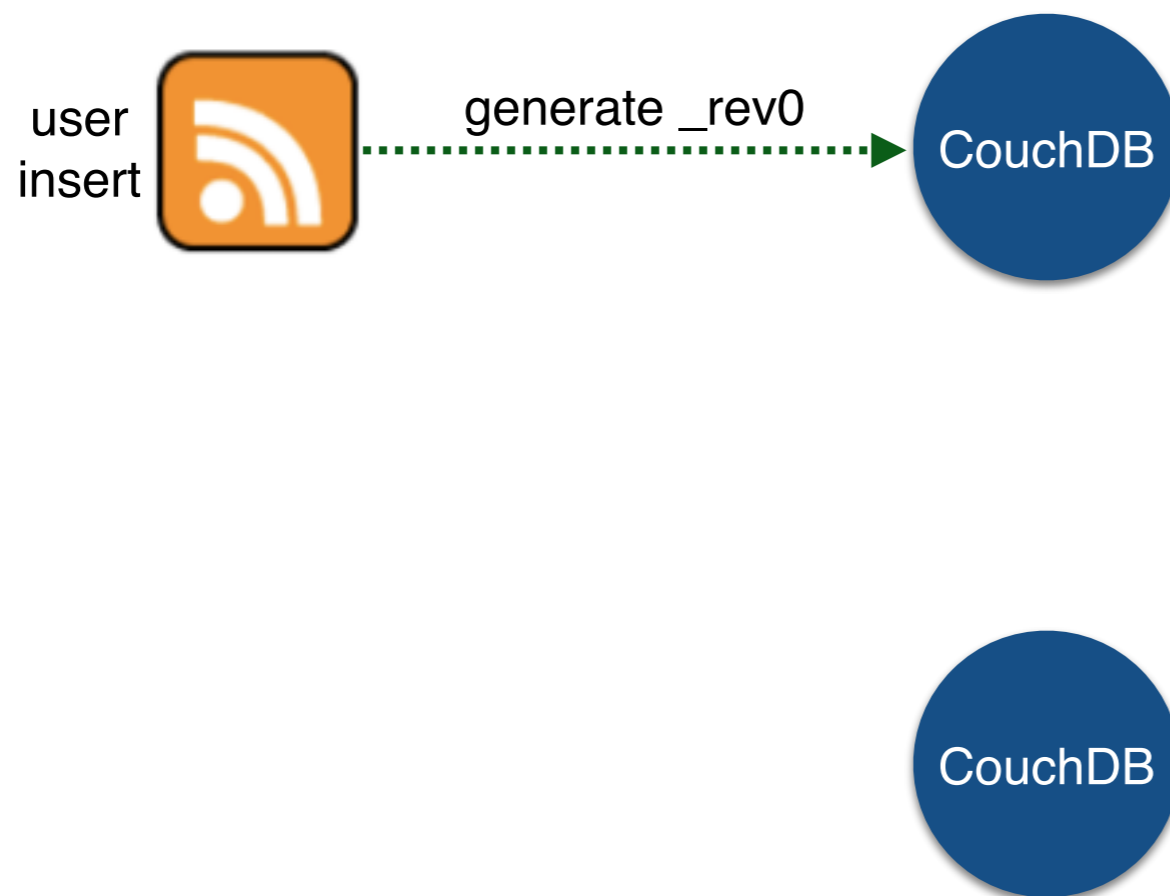
- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary



# Distributed Architecture of CouchDB

---

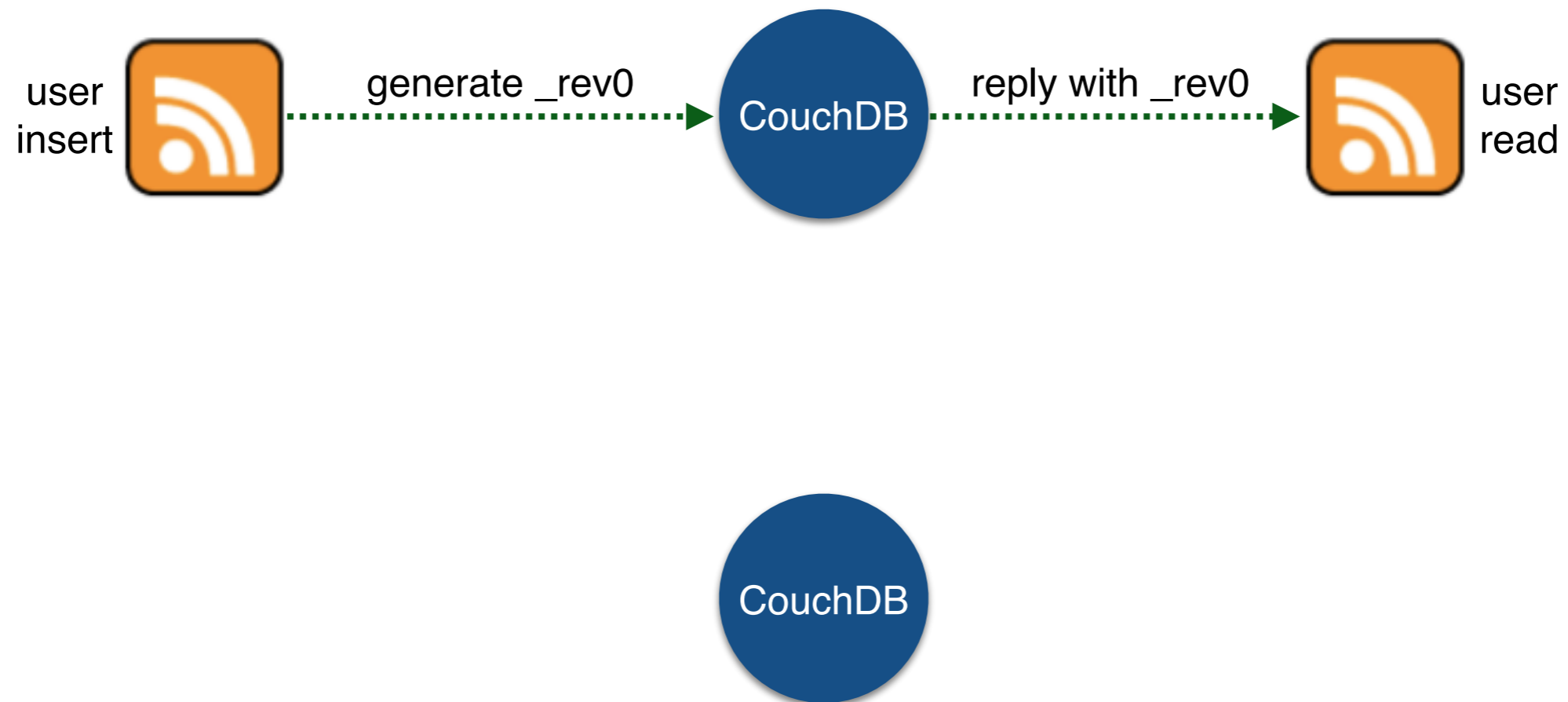
- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary
  - Sites can go offline, DB will handle sync when back online



# Distributed Architecture of CouchDB

---

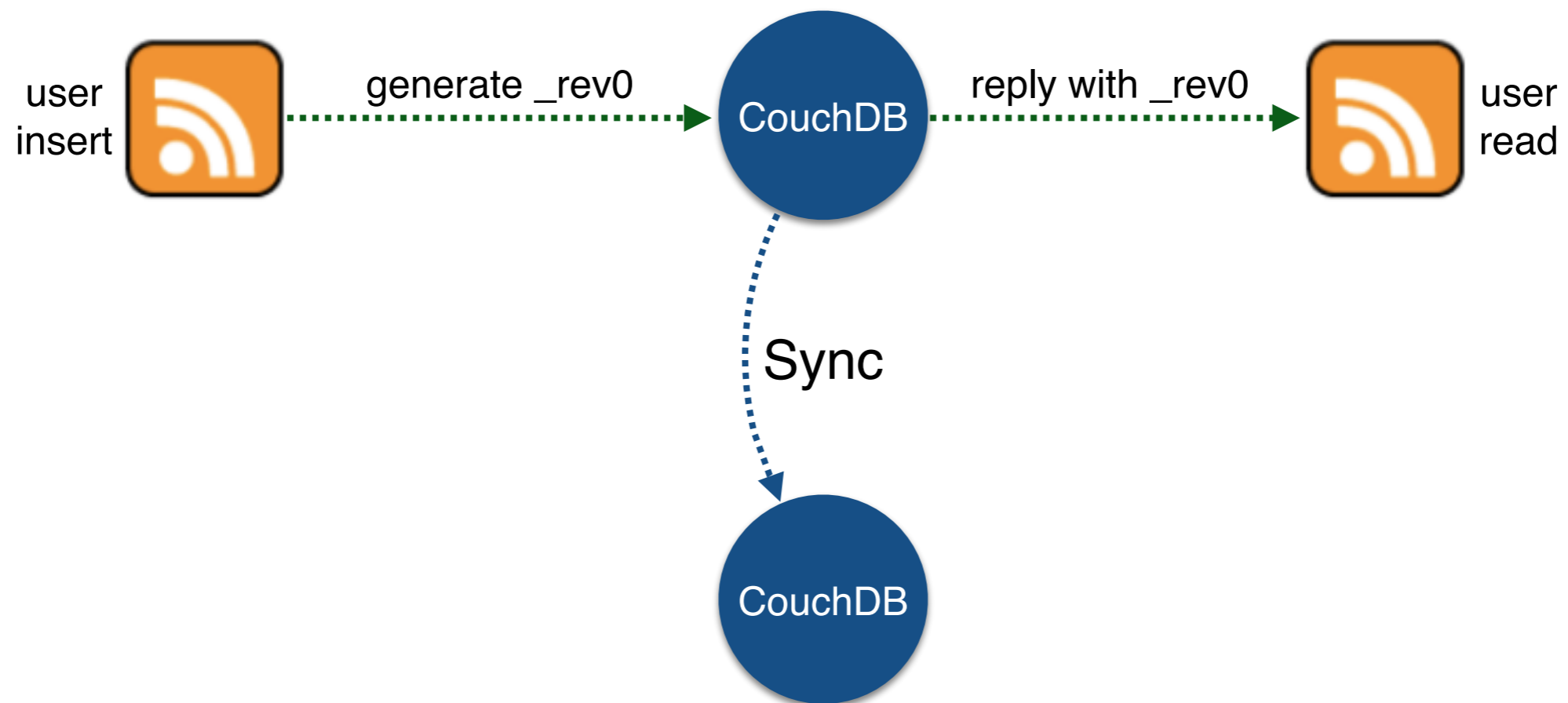
- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary
  - Sites can go offline, DB will handle sync when back online



# Distributed Architecture of CouchDB

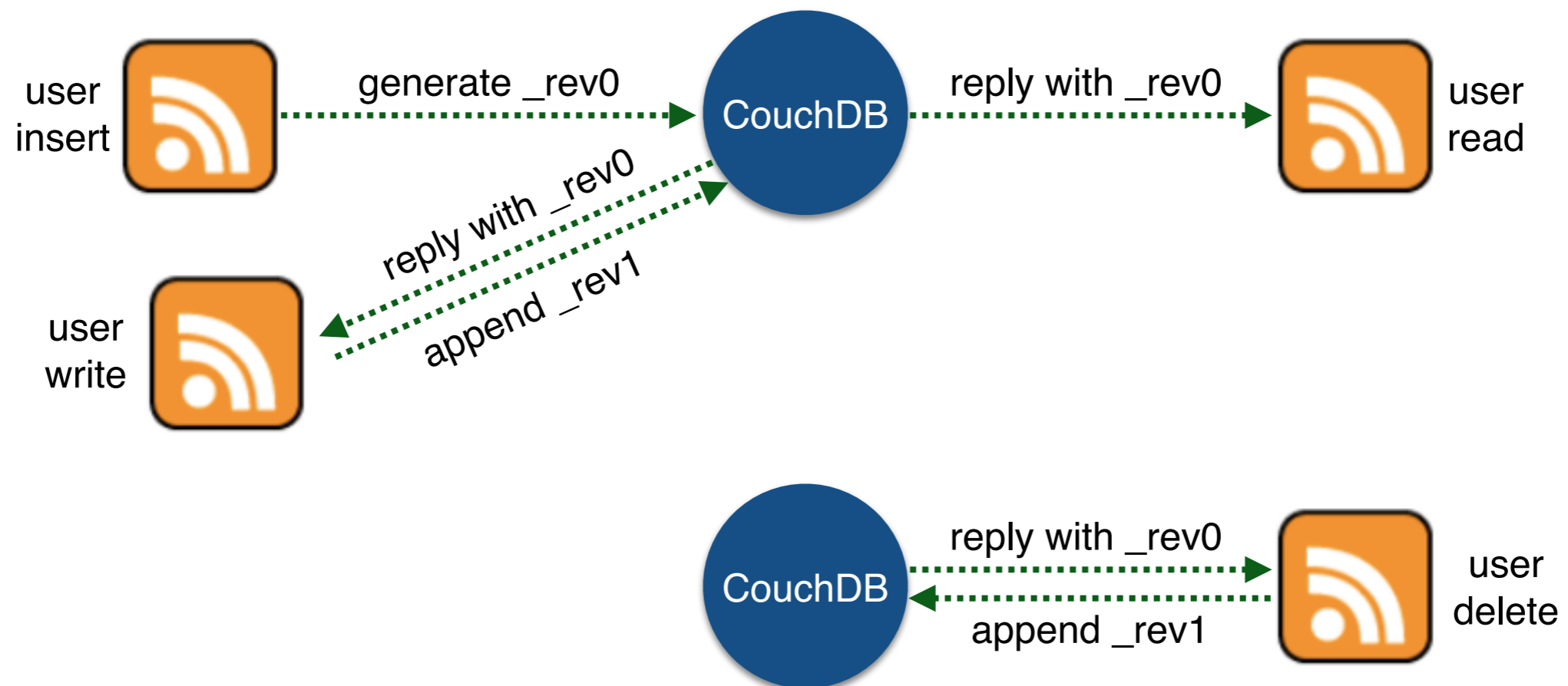
---

- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary
  - Sites can go offline, DB will handle sync when back online



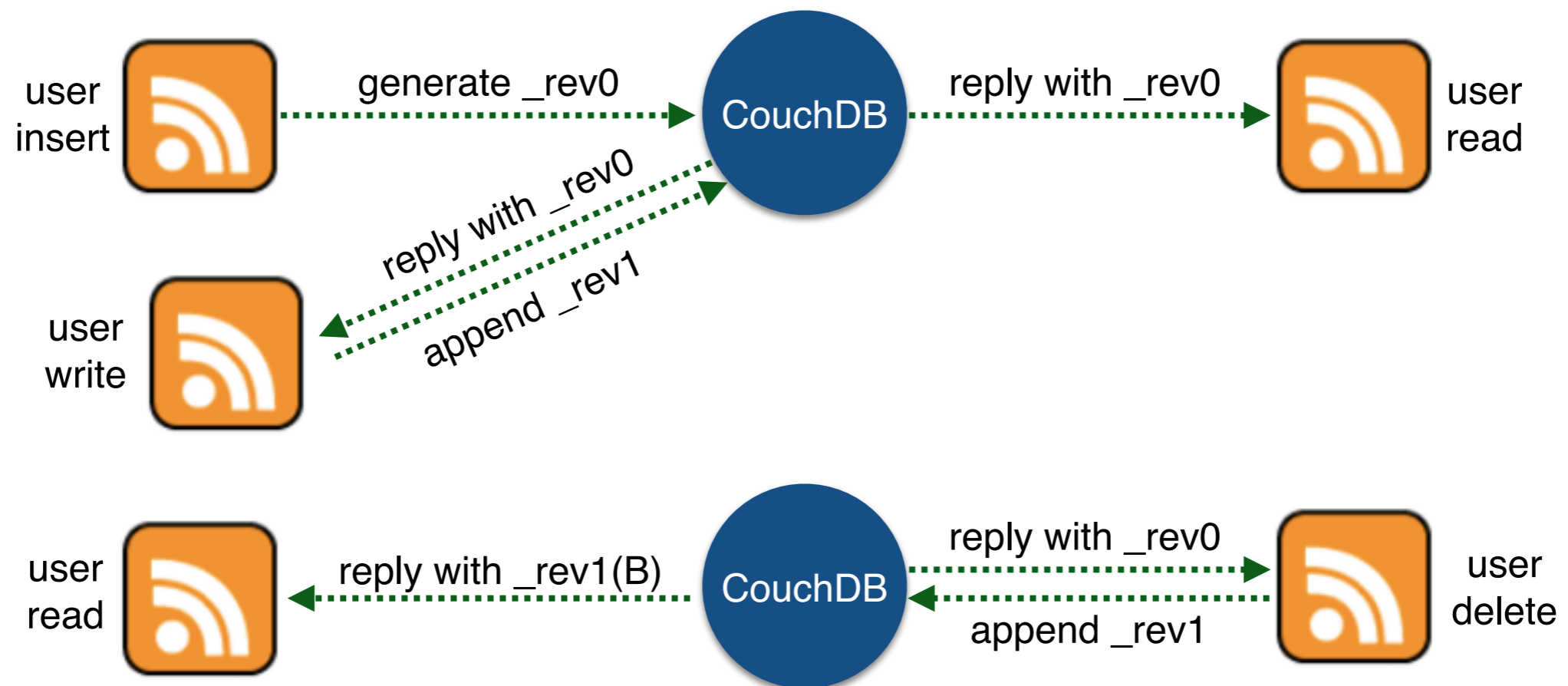
# Distributed Architecture of CouchDB

- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary
  - Sites can go offline, DB will handle sync when back online
  - Automatic conflict detection and resolution



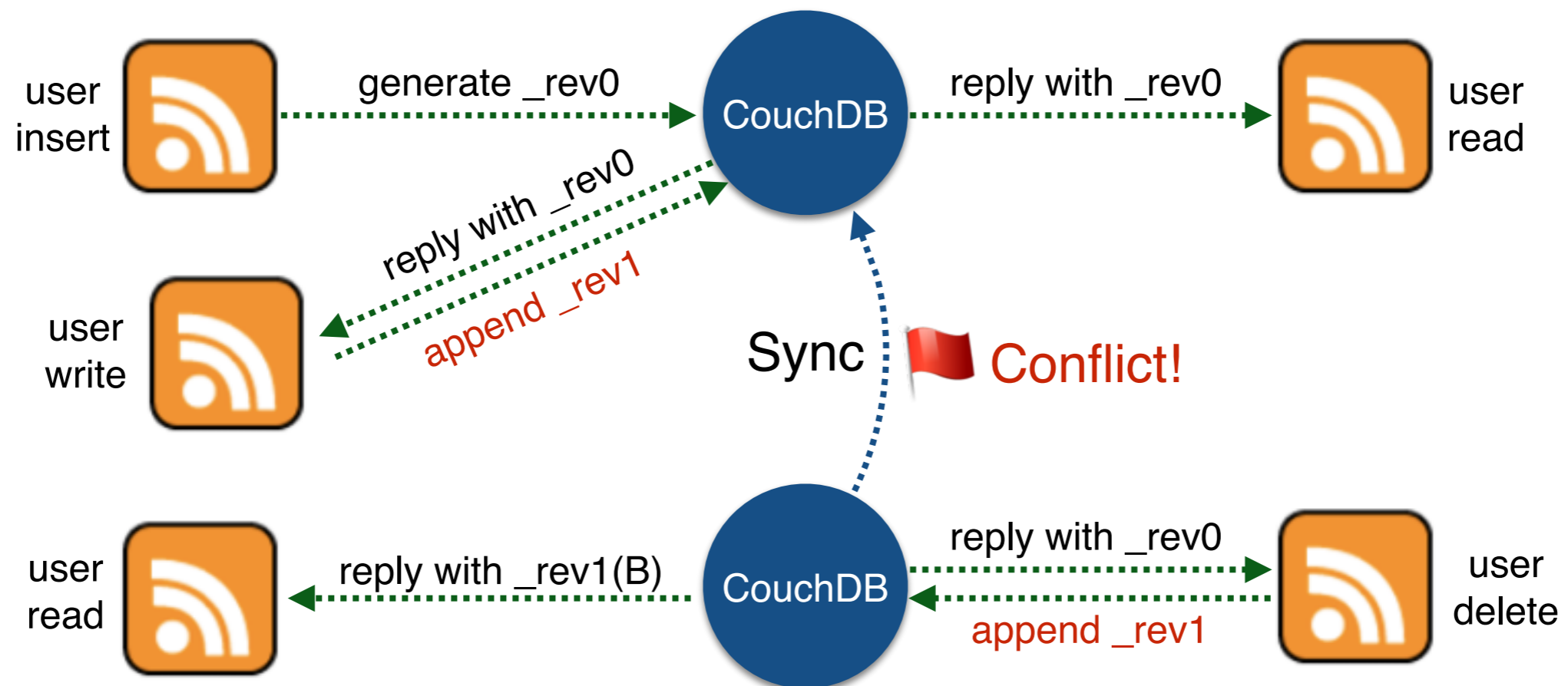
# Distributed Architecture of CouchDB

- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary
  - Sites can go offline, DB will handle sync when back online
  - Automatic conflict detection and resolution



# Distributed Architecture of CouchDB

- Eventual consistency by incremental replication:
  - Peer-to-peer rather than primary-secondary
  - Sites can go offline, DB will handle sync when back online
  - Automatic conflict detection and resolution



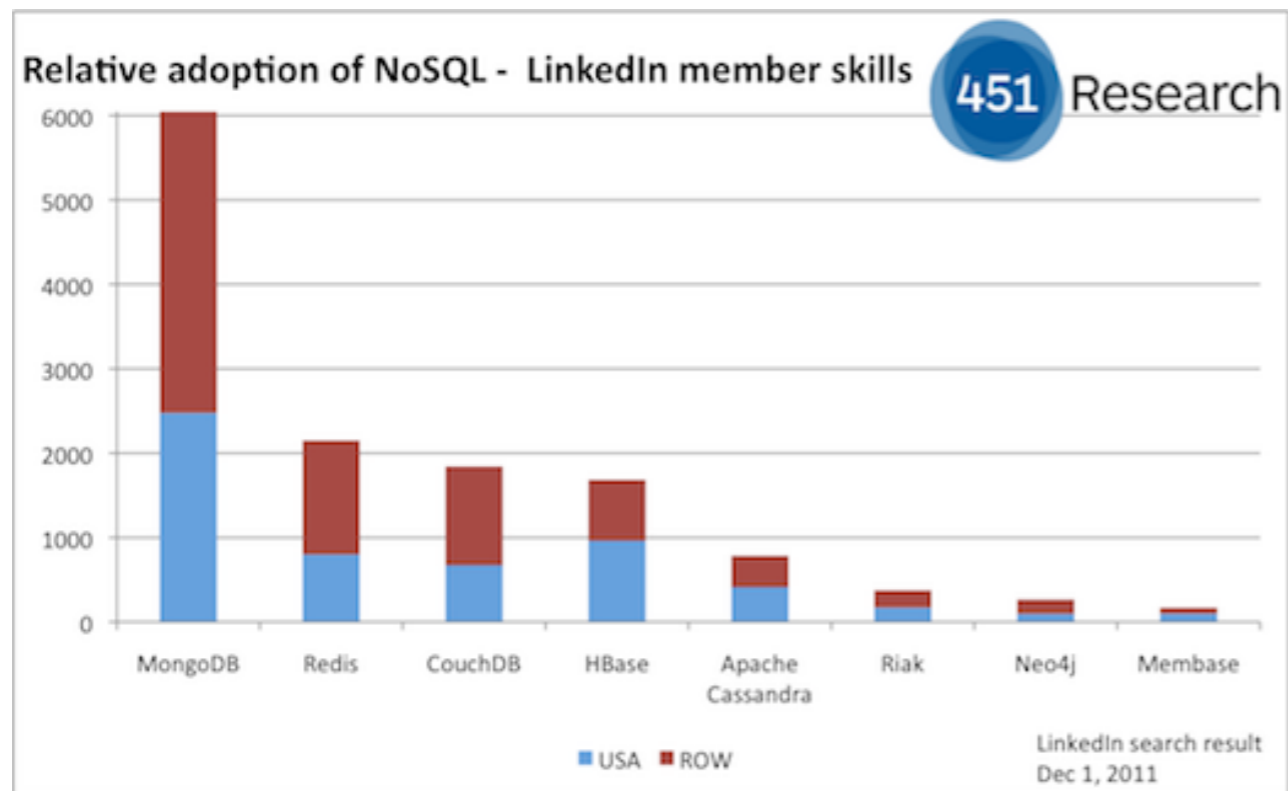


# Conclusions

---

	MongoDB	CouchDB
Focus	Consistency	Availability
Distributed architecture	Primary-Secondary replication	Peer-Peer synchronization
Concurrency control	Update in-place (much like SQL)	MVCC
Document format	BSON	JSON
Data storage	Referenced or embedded	Self-contained
Data organization	One extra layer: collections	Everything piled together
Query capabilities	Aggregation pipeline or MapReduce	MapReduce views and indexes
CRUD syntax	SQL-like	HTTP methods

# Some facts: Popularity

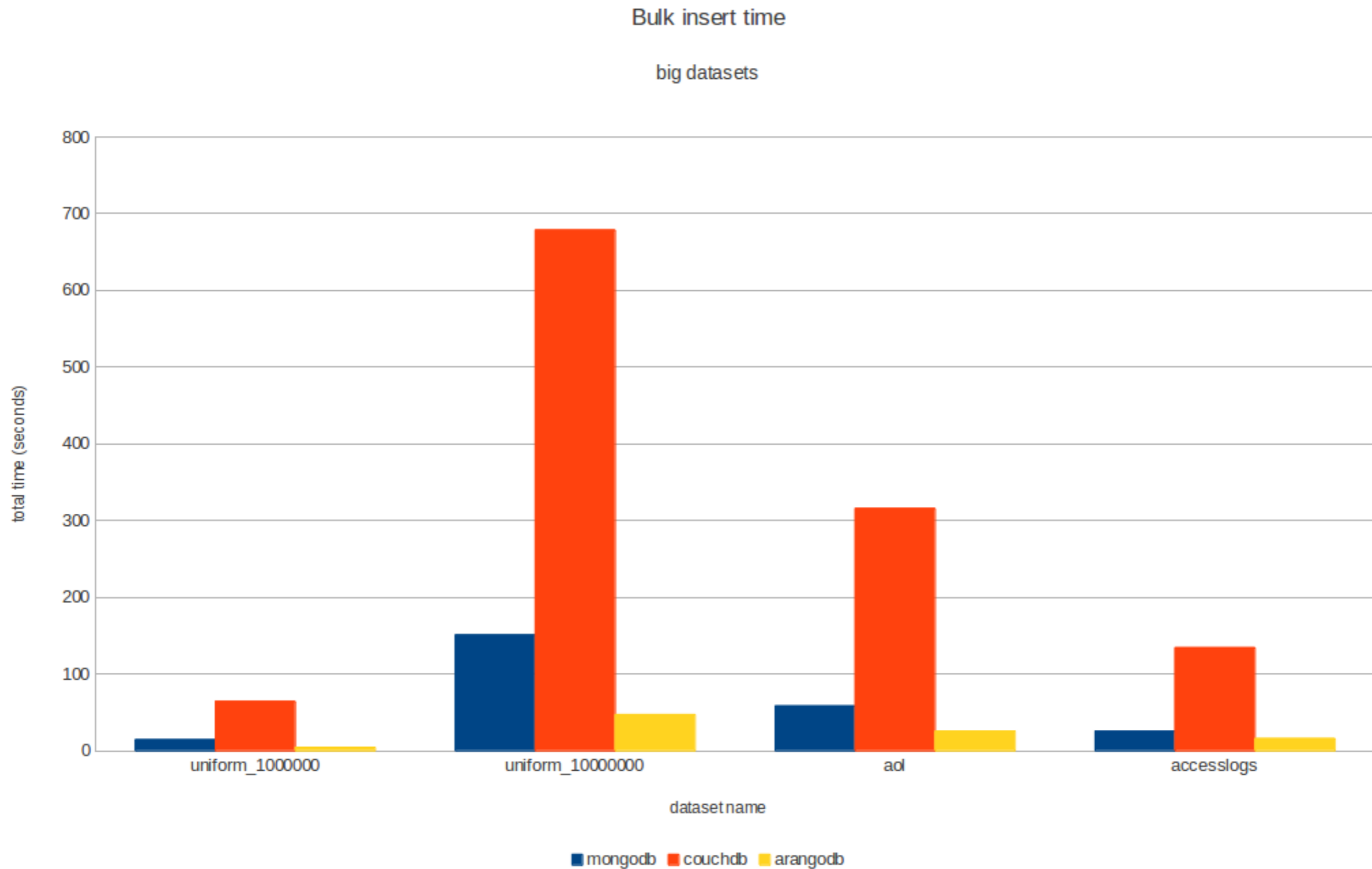


# ubuntu

(Stopped using in 2011...)

# Some facts: Efficiency

---



# When to use what?

---

- You have some predefined queries upfront, want to run on occasionally changing data;
- Need to make sure that sites are always available, even if data center crashes;
- Need to replicate data bi-directionally between 2 or more data centers;
- If versioning is important;
- You are familiar with HTTP but not SQL;
- You are a geek and you believe RDBMS is outdated.



# When to use what?

---

- All other cases when you need a distributed DBMS



# Lucene Index

A rich and powerful full-text search toolbox

# Overview

---

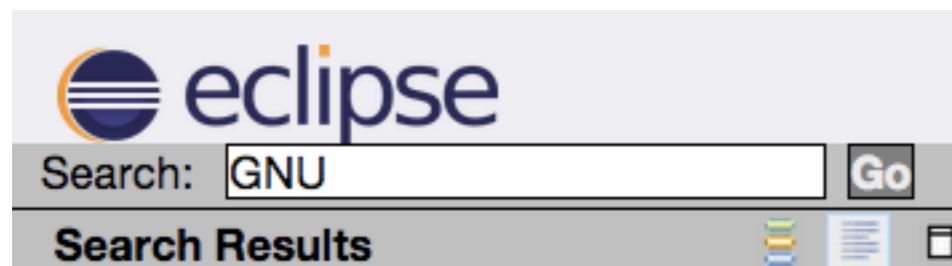
- **Background: Why full text search?**
- **Introduction: What Lucene Index is and is not**
- **Lucene Workflow**
- **Lucene API and Functionalities**
- **Some other useful resources**

# Background: Why full text search?

---

## Question: How do you search for a term in documentation?

- Option 1: Read through 10000 pages of Help documentation
- Option 2: Type the term in the search field



**38 matches in All topics:** [Change scope](#)

 [org.eclipse.cdt.core.dom.ast.gnu \(Eclipse CDT API Specification\)](#)

JavaScript is disabled on your browser.  
Overview Package Class Use Tree  
Deprecated Index Help Eclipse CDT 8.4  
(Luna) Prev Package Next Package  
Frames No Frames All Classes ...

 [org.eclipse.cdt.core.dom.ast.gnu.cpp \(Eclipse CDT API Specification\)](#)

JavaScript is disabled on your browser.  
Overview Package Class Use Tree

Lucky enough.

Eclipse help system supports option 2.

What they use is Lucene Index.



# Background: Why full text search?

---

Question: What if you want to search for a term in a disk of 100G?

- Option 1: grep => couple hours
- Option 2: Create an index => milliseconds

Question: How do you want the result to be reported?

- Option 1: A long list of all matched hits
- Option 2: Aggregated? Sorted? Filtered?



# Background: Why full text search?

---

## 3 Fundamental Questions

- What do you put in the index?
- How do you create the index?
- How do you search the index?

# Introduction: What Lucene Index is and is not

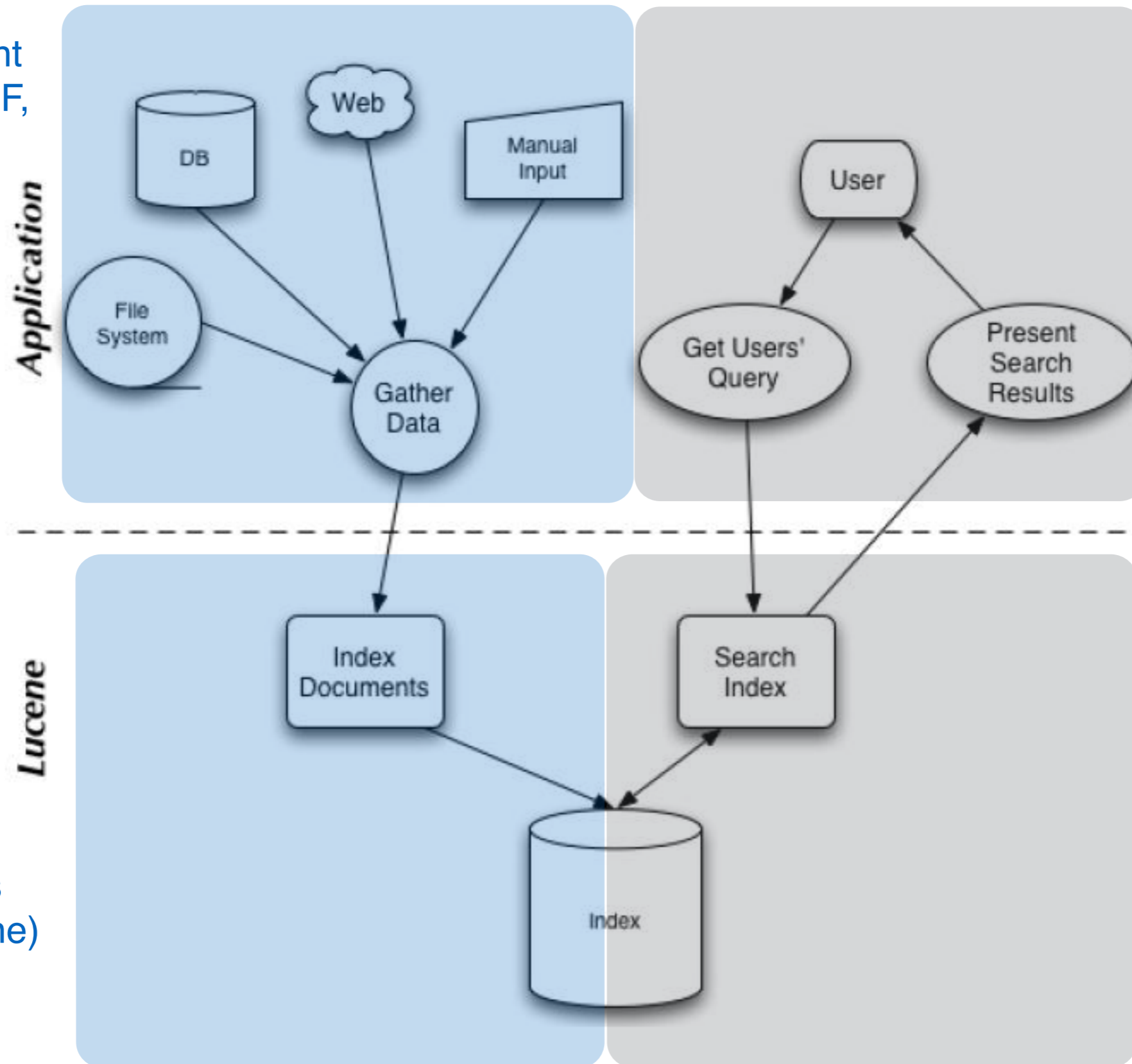
---



- An open source, cross-platform full-text indexing and search library in Java
- Licence: Apache, Original author: Doug Cutting
- Widely recognized for implementing both Internet search engines and local single-site searching
- Lucene is **not** a search engine. It is **not** an application.

# Lucene Index workflow

Flexible document format: HTTP, PDF, metadata, etc.

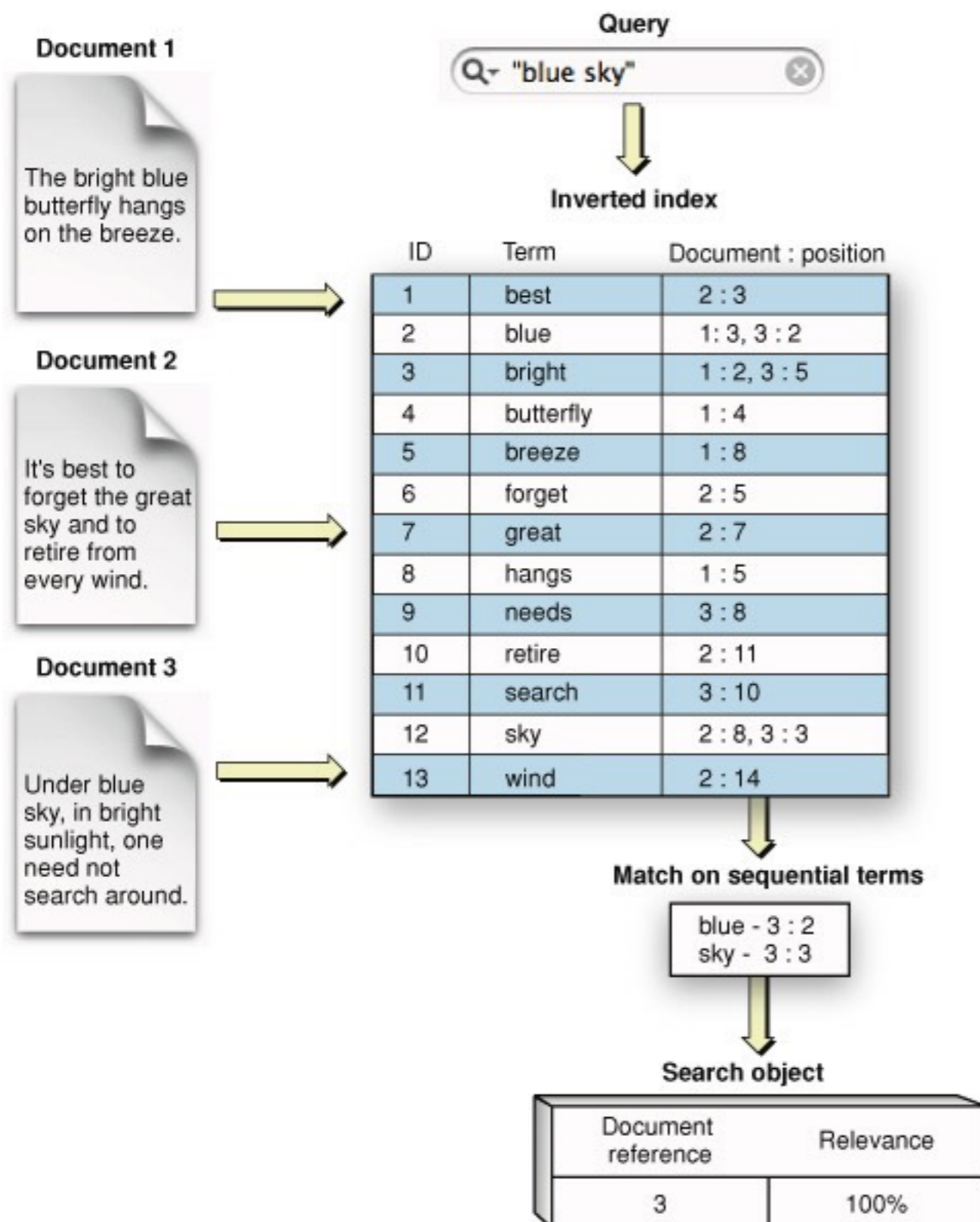


Rich, powerful search algorithms, Easy-to-use API for query

Create index on documents (core of Lucene)

Parse query and answer

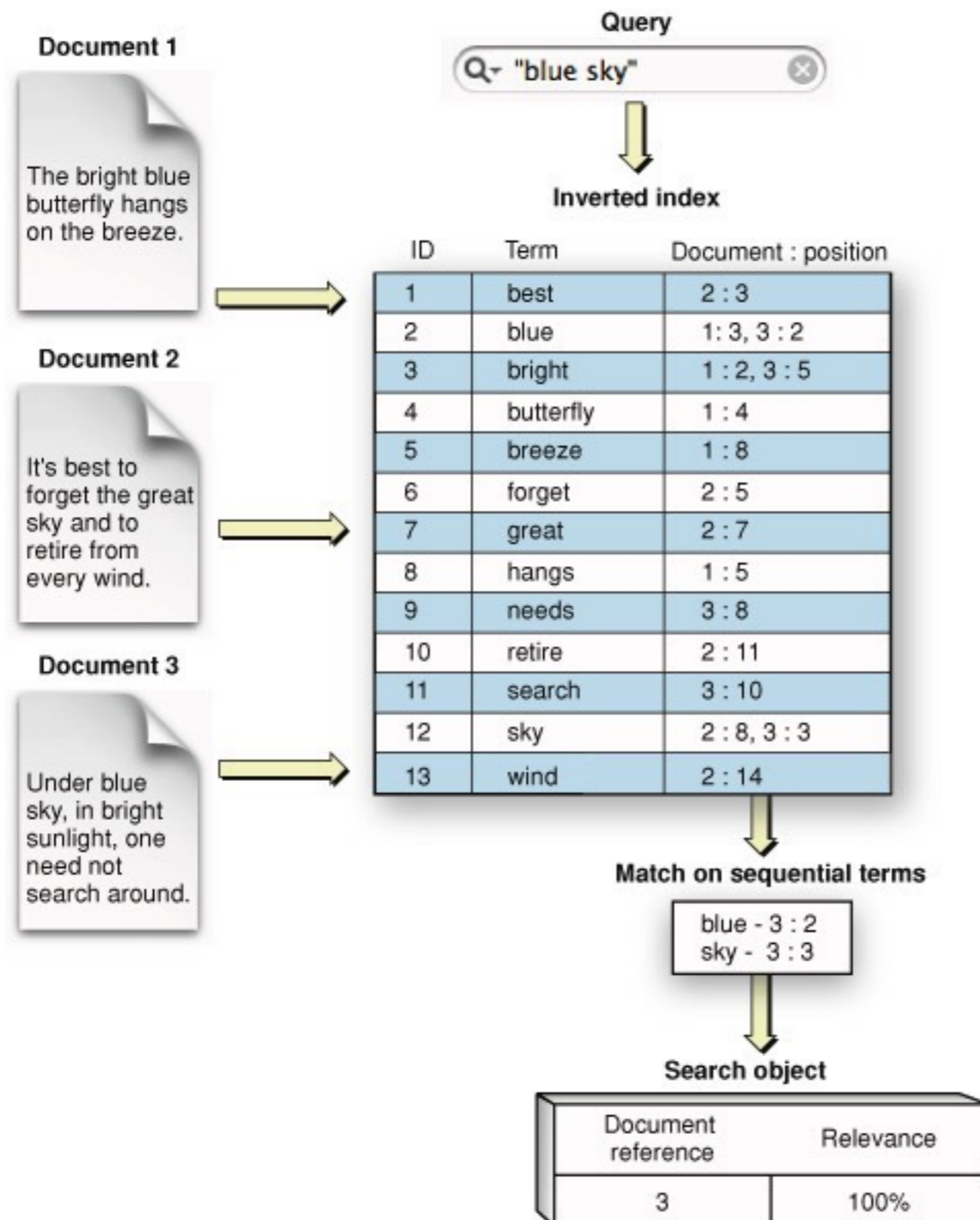
# Indexing Fundamentals



## INDEXING FUNDAMENTALS

- An index is maintained for a collection of documents
- A document is a collection of fields
- A field is a named collection of terms
- A term is a paired string: <field, term-string>
- Inverted index for efficient term-based search

# Lucene API



## STEP 1: CREATE INDEX

- Document: a record.
- Field: features of a record.
- Analyzer: parse each field into indexable tokens
- IndexWriter: create index and add new index entries
- Directory: where index is stored

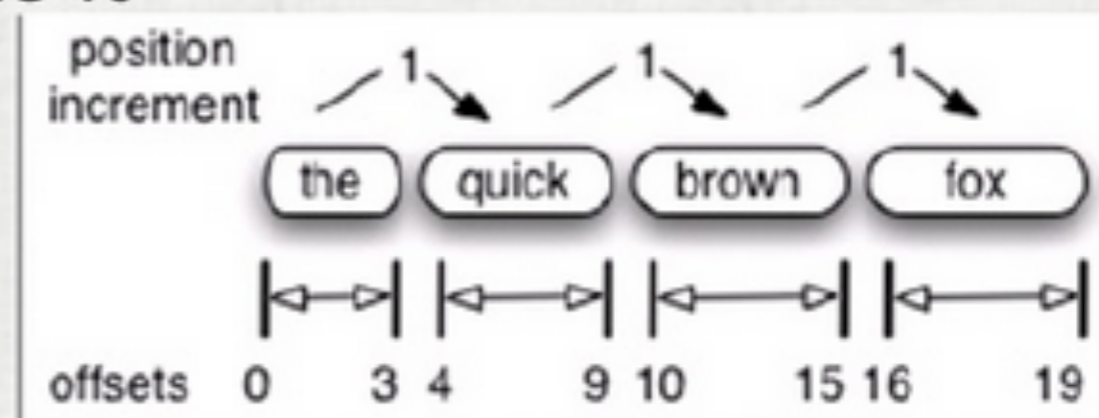
# ANALYZERS

- Process of converting field text into its most fundamental indexed representation, terms.
- Analyzer tokenizes text by performing following tasks:
  - extracting words
  - discarding punctuation, removing accents from characters
  - lowercasing (also called normalizing)
  - removing common words
  - reducing words to a root form (stemming)
  - changing words into the basic form (lemmatization).
- Analysis done at 2 steps
  - Adding fields to index
  - Preparing user query for searching

# ANALYZERS contd..

- Lucene has 4 analyzers built into it

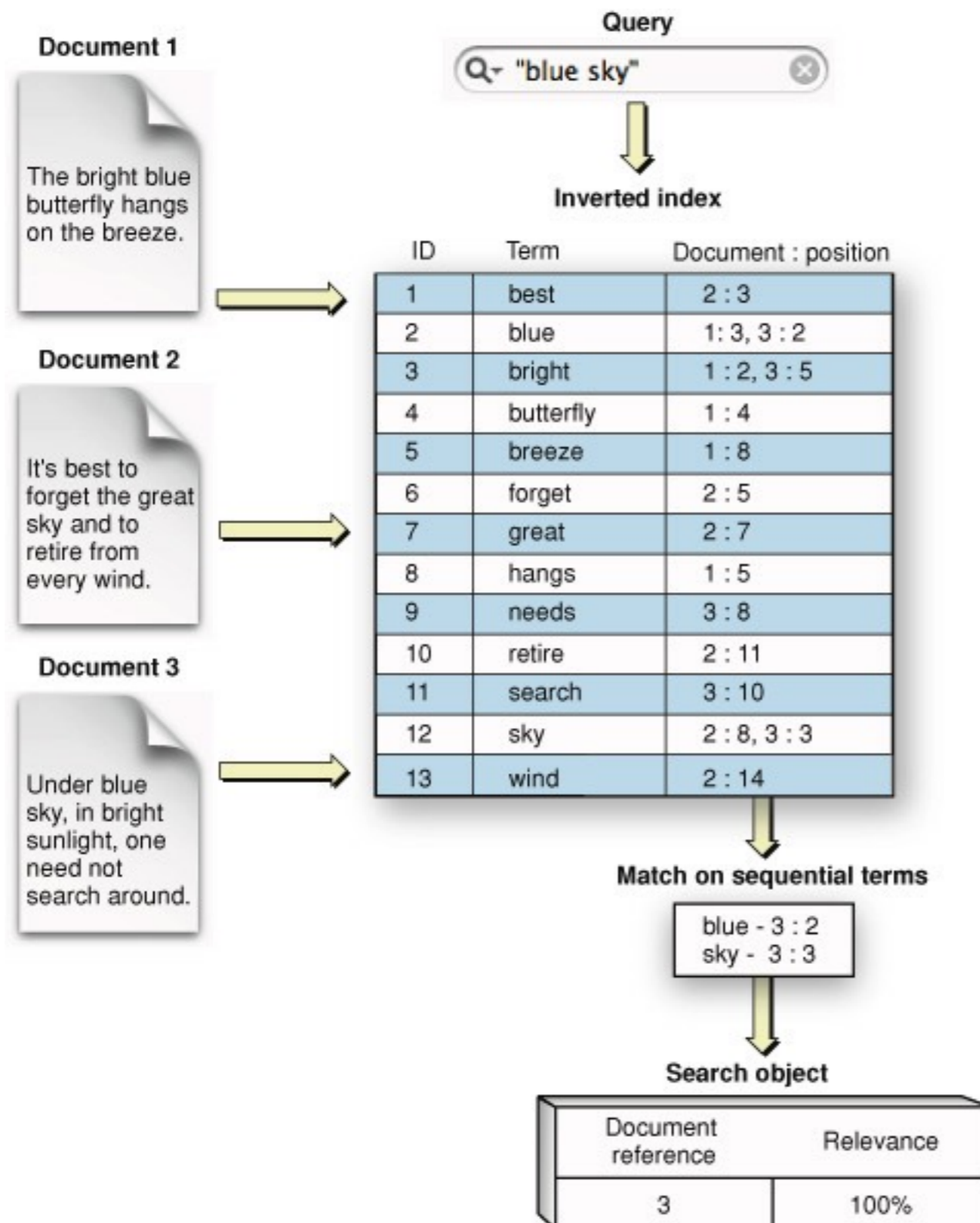
- *Whitespace Analyzer*
- *Simple Analyzer*
- *Stop Analyzer*
- *Standard Analyzer*



- A stream of tokens is the fundamental output of the analysis process.
- During indexing, fields designated for tokenization are processed with the specified analyzer, and each token is written to the index as a term.
- Analyzers don't help in field separation because their scope is to deal with a single field at a time. Instead, parsing these documents prior to analysis is required.



# Lucene API



## STEP 2: QUERY INDEX

- Query: parse query string
- Term: basic unit for search
- TermQuery: subclass for query operation
- IndexSearcher: search in a built index
- Hits: store search results

# SEARCHING

- Lucene provides a powerful Search syntax.
- Supports several kinds of advanced searches.
  - Boolean operators – AND, OR, NOT
  - Field search - "title:Lucene AND content:Java"
  - Wildcard search - "tex\*", "tex?", "?ex\*"
  - Fuzzy search – "Solarus~"
  - Range search – "birthday [20000101 – 20060606]"

# Lucene scoring

$$\sum_{t \text{ in } q} tf(t \text{ in } d) \cdot idf(t) \cdot boost(t.\text{field in } d) \cdot lengthNorm(t.\text{field in } d)$$

Factor	Description
tf(t in d)	Term frequency factor for the term (t) in the document (d).
idf(t)	Inverse document frequency of the term.
boost(t.field in d)	Field boost, as set during indexing.
lengthNorm(t.field in d)	Normalization value of a field, given the number of terms within the field. This value is computed during indexing and stored in the index.
coord(q, d)	Coordination factor, based on the number of query terms the document contains.
queryNorm(q)	Normalization value for a query, given the sum of the squared weights of each of the query terms.

# Lucene algorithms

---

## Lucene Indexing Algorithm

- K-way merge – process at disk transfer rate
- Average  $b \cdot \log N$  indexes
  - $n=1M, b=2$  gives 20 indexes
  - Fast to update and not too slow to search
- Optimization
  - Small sized segments kept in RAM, saves I/O calls

## Search Algorithm

- Posting Lists (from .frq file)
  - For every term  $T_i$ , we have a posting list
    - $T_i \rightarrow (\text{Doc-id}, \text{Freq}(d, t)) *$
    - $T_i$ 's are sorted lexically, so are posting list on doc-id
  - Postings are delta encoded (for Index Compression)
- Based on Vector Space Model
  - Conjunctive Search Algorithm (AND queries)
  - Disjunctive Search Algorithm (OR queries)

# Integration of CouchDB and Lucene Index

---

An open source project for full-text indexing CouchDB:

<https://github.com/rnewson/couchdb-lucene>

(Outperforms MongoDB this time!)

# REFERENCES

---

## CouchDB:

- CouchDB: The Definitive Guide
- <http://couchdb.apache.org/>
- <https://highlyscalable.wordpress.com/2012/03/01/nosql-data-modeling-techniques/>
- <http://blog.scottlogic.com/2014/08/04/mongodb-vs-couchdb.html>
- <http://openmymind.net/2011/10/27/A-MongoDB-Guy-Learns-CouchDB/>

## Lucene Index:

- <https://lucene.apache.org/core/documentation.html>
- <http://www.cnblogs.com/forfuture1978/archive/2010/04/04/1704282.html>