Spark
Consistency at Scale
NoSQL

Chris Rossbach
cs378 Fall 2018
11/28/18
Pro Forma

• Questions?
• Please fill out the eCIS survey!
• Concurrency at scale
  • Spark
  • NoSQL
  • Consistency

• Acknowledgements:
  • https://cseweb.ucsd.edu/classes/wi17/cse291-d/applications/In/NoSQLMongo.pptx
  • http://grimstad.uia.no/janpn/IKT437/2016/slides/ppt/NoSQL-2016.pptx
  • https://www.intertech.com/resource/usergroup/NoSQL.ppt
  • https://courses.engr.illinois.edu/cs525/sp2017/L6.sp17.pptx
  • https://www.cs.princeton.edu/~mfreed/docs/replex-atc16-slides.pptx
Consistency at Scale/NoSQL faux quiz:

• What is the CAP theorem? What does “PACELC” stand for; how does it relate to CAP?
• What is the difference between ACID and BASE?
• What is causal consistency?
• What is chain replication?
• Why do NoSQL systems claim to be more horizontally scalable than RDBMSes? List some features NoSQL systems give up toward this goal.
• What is eventual consistency? Give a concrete example of how of why it causes a complex programming model (relative to a strongly consistent model).
• Compare and contrast Key-Value, Document, and Wide-column Stores
• Define and contrast the following consistency properties:
  • strong consistency, eventual consistency, consistent prefix, monotonic reads, read-my-writes, bounded staleness
Review of Data-Parallel Computation Systems

---

**Application**

```haskell
lines = LOAD '/user/hadoop/HDFS_File.txt' AS (line:chararray);
words = FOREACH lines GENERATE FLATTEN(TOKENIZE(line)) as word;
grouped = GROUP words BY word;
wordcount = FOREACH grouped GENERATE group, COUNT(words);
DUMP wordcount;
```

---

**Language**

- SQL
- Sawzall
- Pig, Hive
- LINQ, SQL
- DryadLINQ
- Dryad
- Hadoop
- HDFS
- S3
- Pig, Hive
- Spark
- HPC
- Azure

**Execution**

- Spark
- Hadoop
- HDFS
- S3
- Azure
- SQL Server
- BigTable
- Cosmos
- Cosmos
- Sawzall
- SQL
- ≈SQL
- LINQ, SQL

---

```haskell
count: table sum of int;
total: table sum of float;
sum_of_squares: table sum of float;
x: float = input;
emit count <- 1;
emit total <- x;
emit sum_of_squares <- x * x;
```
MapReduce vs Successors (Dryad, DryadLINQ, Spark)

- DAG instead of BSP
- Diverse Communication Interfaces
  - Memory FIFO
  - Disk
  - Network
  - (MR had just file system)
- Flexible Modular Composition
Dryad (2007): 2-D Pipelines

• Unix Pipes: 1-D
  `grep | sed | sort | awk | perl`

• Dryad: 2-D
  `grep^{1000} | sed^{500} | sort^{1000} | awk^{500} | perl^{50}`
A Dataflow Engine

LINQ / RDD Xform

Compiler

cluster graph

TCP, caches, files

Machine Runtime

Task

A

B

C

D

Machine Runtime

Cluster Runtime

M = Master
S = Slave
CPU = CPU
Virtualizing Dataflow Graphs
Virtualizing Dataflow Graphs
Virtualizing Dataflow Graphs
Virtualizing Dataflow Graphs
Virtualizing Dataflow Graphs
Dataflow Job Structure

Channels

Input files

Vertices (processes)

Stage

Output files

How to implement?
Channels

Finite streams of items

- distributed filesystem files (persistent)
- SMB/NTFS/HDFS files (temporary)
- TCP pipes (inter-machine)
- memory FIFOs (intra-machine)
Dataflow Challenges

• Dataflow is a great execution model
• Dataflow may be less great as a programming model
  • Graph abstraction counter-intuitive
  • Asynchronous execution model
    • Hard to understand
    • Near impossible to debug
• Solution: change programming model
  • Distributed data set: core abstraction
  • Spark, DryadLINQ, Dandelion...
LINQ = .Net+ Queries

```csharp
Collection<T> collection;
bool IsLegal(Key);
string Hash(Key);

var results = from c in collection
              where IsLegal(c.key)
              select new { Hash(c.key), c.value};
```
DryadLINQ (2008) Data Model

Partition

.Net objects

Collection
DryadLINQ = LINQ + Dryad

```csharp
Collection<T> collection;
bool IsLegal(Key k);
string Hash(Key);

var results = from c in collection
    where IsLegal(c.key)
    select new { Hash(c.key), c.value);
```
Language Summary

Where
Select
GroupBy
OrderBy
Aggregate
Join
Apply
Materialize
public static IQueryable<S> MapReduce<T, M, K, S>(
    this IQueryable<T> input,
    Func<T, IEnumerable<M>> mapper,
    Func<M, K> keySelector,
    Func<IGrouping<K, M>, S> reducer)
{
    var map = input.SelectMany(mapper);
    var group = map.GroupBy(keySelector);
    var result = group.Select(reducer);
    return result;
}
Spark (2012)

- Cluster programming models (e.g. MR) de facto standard
  - Fault tolerance through replicated durable storage
  - Dataflow is the common theme
- Multi-core servers common
- Iterative applications common
- MR transforms FS inputs to FS outputs

**Benefits of data flow:** runtime decides where tasks run, automatically recovers from failures

**Downside:** stable storage makes it inefficient
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page’s rank to
   \[ \sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
    ranks = links.join(ranks).flatMap {
        (url, (links, rank)) =>
        links.map(dest => (dest, rank/links.size))
    }.reduceByKey(_ + _)
}
```
Iterative Computations: PageRank

1. Start each page with a rank of 1
2. On each iteration, update each page's rank to
   \[ \sum_{i \in \text{neighbors}} \frac{\text{rank}_i}{|\text{neighbors}_i|} \]

```scala
links = // RDD of (url, neighbors) pairs
ranks = // RDD of (url, rank) pairs

for (i <- 1 to ITERATIONS) {
  ranks = links.join(ranks).flatMap {
    (url, (links, rank)) =>
    links.map(dest => (dest, rank/links.size))
  }.reduceByKey(_ + _)
}
```

- Dataflow is a powerful abstraction
- Inefficient for applications that reuse a working set:
  - Iterative algorithms (many in ML)
  - Interactive data mining tools (R, Python)
- Spark makes working sets a first-class concept

**Solution:** augment data flow model with “resilient distributed datasets” (RDDs)
Spark Programming Model

• Resilient distributed datasets (RDDs)
  • Immutable collections partitioned across cluster that can be rebuilt if a partition is lost
  • Created by transforming data in stable storage using data flow operators (map, filter, group-by, ...)
  • Can be cached across parallel operations

• Parallel operations on RDDs
  • Reduce, collect, count, save, ...

• Restricted shared variables
  • Accumulators, broadcast variables
Example: Log Mining

• Load error messages from a log into memory, then interactively search for various patterns

```scala
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
messages = errors.map(_.split(\t')(2))
cachedMsgs = messages.cache()
cachedMsgs.filter(_.contains("foo")).count
cachedMsgs.filter(_.contains("bar")).count
...
```

**Result:** full-text search of Wikipedia in <1 sec (vs 20 sec for on-disk data)**
## RDD Operations

<table>
<thead>
<tr>
<th>Transformations (define a new RDD)</th>
<th>Parallel operations (return a result to driver)</th>
</tr>
</thead>
<tbody>
<tr>
<td>map</td>
<td>reduce</td>
</tr>
<tr>
<td>filter</td>
<td>collect</td>
</tr>
<tr>
<td>sample</td>
<td>count</td>
</tr>
<tr>
<td>union</td>
<td>save</td>
</tr>
<tr>
<td>groupByKey</td>
<td>lookupKey</td>
</tr>
<tr>
<td>reduceByKey</td>
<td>...</td>
</tr>
<tr>
<td>join</td>
<td></td>
</tr>
<tr>
<td>persist/cache</td>
<td></td>
</tr>
</tbody>
</table>

### Diagram:
- **Where**
- **Select**
- **GroupBy**
- **OrderBy**
- **Aggregate**
- **Join**
- **Apply**
- **Materialize**
RDD Fault Tolerance

- RDDs maintain *lineage* information that can be used to reconstruct lost partitions

- Ex:
  
  ```scala
  cachedMsgs = textFile(...).filter(_.contains("error"))
  .map(_.split('t')(2))
  .persist()
  ```

- Consistency simplified by immutability
- Inexpensive fault tolerance
  - log lineage rather than replicating/checkpointing data
- Locality-aware scheduling of tasks on partitions
- Restricted model, but applicable to important apps
### RDDs vs Distributed Shared Memory

<table>
<thead>
<tr>
<th>Concern</th>
<th>RDDs</th>
<th>Distr. Shared Mem.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reads</td>
<td>Fine-grained</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Writes</td>
<td>Bulk transformations</td>
<td>Fine-grained</td>
</tr>
<tr>
<td>Consistency</td>
<td>Trivial (immutable)</td>
<td>Up to app / runtime</td>
</tr>
<tr>
<td>Fault recovery</td>
<td>Fine-grained and low-overhead using lineage</td>
<td>Requires checkpoints and program rollback</td>
</tr>
<tr>
<td>Straggler mitigation</td>
<td>Possible using speculative execution</td>
<td>Difficult</td>
</tr>
<tr>
<td>Work placement</td>
<td>Automatic based on data locality</td>
<td>Up to app (but runtime aims for transparency)</td>
</tr>
</tbody>
</table>
Dataflow Engines Summary

• Manage concurrency by restricting programming model

• Dataflow execution model very powerful
  • Important concerns handled transparently by runtime
    • Scheduling
    • Data partitioning and movement
    • Synchronization / Coordination
  • Challenge: what is the right programming model for dataflow?

• MapReduce—provide just two (Turing complete) functions

• DryadLINQ/Spark: encapsulate concurrency and fault tolerance behind data abstractions
What is NoSQL?

- Next Generation Compute/Storage engines (databases)
  - non-relational
  - distributed
  - open-source
  - horizontally scalable
- One view: “no” → elide SQL/database functionality to achieve scale
- Another view: “NoSQL” is actually misleading.
  - more appropriate term is actually “Not Only SQL”

What NoSQL gives up in exchange for scale:

Why talk about NoSQL in concurrency class?

- Principle
  - Most tradeoffs are a direct result of concurrency
- Practice
  - NoSQL systems are ubiquitous
- Relevant aspects
  - scale/performance tradeoff space
  - Correctness/programmability tradeoff space
RDBMS Vs. NoSQL

RDBMS
- Structured and organized data
- Structured Query Language (SQL)
- Data and its relationships stored in separate tables.
- Data Manipulation Language, Data Definition Language
- Strong Consistency
- Transactions

NoSQL
- No declarative query language
- No predefined schema
- Key-Value pair storage, Column Store, Document Store, Graph Databases
- Eventual consistency rather ACID property (BASE)
- Unstructured and unpredictable data

All true…but
- This is a BORING way to understand NoSQL
- And is of limited utility
A Better Framework

Still not a perfect framework

Cons:
- Many dimensions contain sub-dimensions
- Many concerns fundamentally coupled
- Dimensions are often un- or partially-ordered

Pros:
- Makes important concerns explicit
- Cleanly taxonomizes most modern systems

Key Value Stores
- Basics Available
- Soft State
- Eventually Consistent

Document Stores
- Shared-Disk
- Range-Sharding
- Primary-Backup
- Commit-Consensus

Table-Column Stores
- Logging
- Secondary Indexing
- Query Planning
- Materialized Views
- Analytics

Replication
- Atomicity
- Consistency
- Isolation
- Durability

Storage
- In-Memory Storage
- Update In Place
- Caching
- Secondary Indexing
- Query Planning
- Materialized Views
- Analytics

Query Support
- Sync/Async
- Logging
- Update In Place
- Caching
- Secondary Indexing
- Query Planning
- Materialized Views
- Analytics
Consistency

How to keep data in sync

- Partitioning → single row spread over multiple machines
- Redundancy → single datum spread over multiple machines
Consistency: the problem

- Clients perform reads and writes
- Data is replicated among a set of servers
- Writes must be performed at all servers
- Reads return the result of one or more past writes

![Diagram showing a writer writing to two servers (R1, R2), and a reader reading from one of them.](image)

- How should we *implement* write?
- How to *implement* read?
Consistency: CAP Theorem

• A distributed system can satisfy at most 2/3 guarantees of:

  1. **Consistency**:
     • all nodes see same data at any time
     • or reads return latest written value by any client

  2. **Availability**:
     • system allows operations all the time,
     • and operations return quickly

  3. **Partition-tolerance**:
     • system continues to work in spite of network partitions
Why Care about CAP Properties?

Availability
- Reads/writes complete reliably and quickly.
- At Amazon, each added ms of latency implies a $6M yearly loss.

Partitions
- Internet router outages
- Under-sea cables cut
- DNS not working
- rack switch outage
- system should continue functioning normally!

Consistency
- all nodes see same data at any time, or reads return latest written value by any client.
- This basically means correctness!
CAP Implications

- A distributed storage system can achieve at most two of C, A, and P.

- When partition-tolerance is important, you have to choose between consistency and availability.

PACELC:

```java
if(partition) {
    choose A or C
} else {
    choose latency or consistency
}
```
Consistency Spectrum

Faster reads and writes

More consistency

Eventual

Strong (e.g., Sequential)
Spectrum Ends: Eventual Consistency

- **Eventual Consistency**
  - If writes to a key stop, all replicas of key will converge
  - Originally from Amazon’s Dynamo and LinkedIn’s Voldemort systems
Spectrum Ends: Strong Consistency

• **Linearizability:**
  • Each operation is visible (or available) **instantaneously** to all other clients

• **Sequential Consistency** [Lamport]:
  • "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
  • After the fact, find a “reasonable” ordering of the operations (can re-order operations) that obeys sanity (consistency) at all clients, and across clients.

• **ACID** properties
Many Many Consistency Models

- Amazon S3 – eventual consistency
- Amazon Simple DB – eventual or strong
- Google App Engine – strong or eventual
- Yahoo! PNUTS – eventual or strong
- Windows Azure Storage – strong (or eventual)
- Cassandra – eventual or strong (if R+W > N)
- ...

**Question:** How to choose what to use or support?
# Some Read Consistency Guarantees

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
</tr>
</tbody>
</table>
Comparing Consistency

- Strong
- Prefix
- Bounded
- Monotonic
- RMW
- Eventual

strength

metric = set of allowable read results
## Consistency Trade-offs

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>See Description</th>
<th>Consistency</th>
<th>Performance</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
<td>A</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
<td>D</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
<td>C</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>
The Game of Soccer

for half = 1 .. 2 {
    while half not over {
        kick-the-ball-at-the-goal
        for each goal {
            if visiting-team-scored {
                score = Read ("visitors");
                Write ("visitors", score + 1);
            } else {
                score = Read ("home");
                Write ("home", score + 1);
            }
        }
    }
    hScore = Read("home");
    vScore = Read("visit");
    if (hScore == vScore)
        play-overtime
Desired consistency?

**Strong**

= Read My Writes!
vScore = Read (“visitors”);
hScore = Read (“home”);
if vScore == hScore
    play-overtime

Desired consistency?

Strong consistency

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
</tr>
</tbody>
</table>
Radio Reporter

do {
    BeginTx();
    vScore = Read ("visitors");
    hScore = Read ("home");
    EndTx();
    report vScore and hScore;
    sleep (30 minutes);
}

<table>
<thead>
<tr>
<th>Desired consistency?</th>
<th>Consistent Prefix</th>
<th>Monotonic Reads</th>
<th>or Bounded Staleness</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all &quot;old&quot; writes.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Sportswriter

While not end of game {
  drink beer;
  smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;

Desired consistency?

Eventual

Bounded Staleness

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all &quot;old&quot; writes.</td>
</tr>
</tbody>
</table>
Statistician

Wait for end of game;
score = \textbf{Read} ("home");
stat = \textbf{Read} ("season-goals");
\textbf{Write} ("season-goals", stat + score);

Desired consistency?

\textbf{Strong Consistency} (1st read)

\textbf{Read My Writes} (2\textsuperscript{nd} read)

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all &quot;old&quot; writes.</td>
</tr>
</tbody>
</table>
Stat Watcher

```plaintext
do {
    stat = Read ("season-goals");
    discuss stats with friends;
    sleep (1 day);
}
```

Desired consistency?

**Eventual Consistency**

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
</tr>
</tbody>
</table>
Official scorekeeper:

score = Read ("visitors");
Write ("visitors", score + 1);

Statistician:

Wait for end of game;
score = Read ("home");
stat = Read ("season-goals");
Write ("season-goals", stat + score);

Referee:

Radio reporter:

do {
    vScore = Read ("visitors");
hScore = Read ("home");
    report vScore and hScore;
sleep (30 minutes);
}

Sportswriter:

While not end of game {
    drink beer;
    smoke cigar;
}
go out to dinner;
vScore = Read ("visitors");
hScore = Read ("home");
write article;

Stat watcher:

stat = Read ("season-runs");
discuss stats with friends;
Relational Database vs Key-Value Stores

Example SQL query:
```
SELECT users.zipcode, blog.num_posts
FROM users JOIN blog
ON users.blog_url = blog.url
```

**WHAT DOES THIS QUERY DO?**

- Gets zip code and number of blog posts for each user

**Problems**
- Complexity for simple queries
- Data: Large and unstructured
- Lots of random reads and writes
- Difficult to scale
  - Foreign keys rarely needed
  - Joins infrequent
Key-value/NoSQL Data Model

- API: `get(key), put(key, value)`
- Unstructured
- Columns Missing from some Rows
  - (no foreign key)
- No schema imposed
- Joins may not be supported

```
SELECT userszipcode, blog.num_posts FROM users JOIN blog ON users.blog_url = blog.url
```

So how do we do this now?
Data Model: Document Stores

• Store data as JSON or BSON (Binary JavaScript Object Notation) documents

```json
{
  name: "travis",
  salary: 30000,
  designation: "Computer Scientist",
  teams: [ "front-end", "database" ]
}
```

• Group related documents with a shared common index is a collection

Why is this an improvement?
• Fetch entire documents
• Leverage tools for manipulating documents
• Easier to implement search
• Easier to implement richer queries
Document Store Example

Query all employee names with salary greater than 18000 sorted in ascending order

**SQL:**
```
SELECT name
FROM employees JOIN salaries
WHERE salary > 18000
ORDER BY salary
ASCENDING
```

**K-V Store:**
- Option 1: Create another table
- Option 2: Iterate existing tables in code

```
foreach k1 in employees.Keys()
    v1 = employees.get(k1)

foreach k2 in salaries.Keys()
    v2 = salaries.get(k2)
    if(v2 > 18000)
        results.add(v1.name, v2)

results.sort((a,b) => (a.v2 <= b.v2))
return results
```
Document Store Example

Query all employee names with salary greater than 18000 sorted in ascending order

Tables ➔ document collections

db.employee.insert(
    {
        name: "sally",
        salary: 15000,
        designation: "MTS",
        teams: [ "cluster-management" ]
    }
)
**Document Store Example**

*Query all employee names with salary greater than 18000 sorted in ascending order*

```
db.employee.find({salary: {$gt:18000}, {name:1}}).sort({salary:1})
```

**Collection**
- {salary:25000, ...}
- {salary:10000, ...}
- {salary:20000, ...}
- {salary:2000, ...}
- {salary:30000, ...}
- {salary:21000, ...}
- {salary:5000, ...}
- {salary:50000, ...}

**Condition**
- {salary:25000, ...}
- {salary:20000, ...}
- {salary:30000, ...}
- {salary:21000, ...}
- {salary:25000, ...}
- {salary:30000, ...}
- {salary:21000, ...}
- {salary:25000, ...}

**Projection**
- {salary:20000, ...}
- {salary:21000, ...}
- {salary:25000, ...}
- {salary:30000, ...}
- {salary:50000, ...}

**Modifier**
- {salary:20000, ...}
- {salary:21000, ...}
- {salary:25000, ...}
- {salary:30000, ...}
- {salary:50000, ...}

- Documents replace tables
- Richer queries
- Still less structure than SQL
### Wide Column Store Motivation

<table>
<thead>
<tr>
<th>col0</th>
<th>col1</th>
<th>col2</th>
<th>…</th>
<th>coln</th>
</tr>
</thead>
<tbody>
<tr>
<td>r0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>r1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>r2</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>…</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>rnc</td>
</tr>
</tbody>
</table>

**Default partitioning key**

- The diagram visualizes the structure of a wide column store, with rows and columns labeled accordingly.
- The default partitioning key is indicated by the highlighted row.
Wide Column Store Motivation

Rows from the same table are dispersed across lots of different machines!
Wide Column Stores: Scale with Partitioning

Store columns together (or a group of columns).

- Entries within a column are indexed
- Contrast: RDBMSs store rows together (on disk or at a server)
- Range searches within a column are fast
- E.g., Get all blog_ids from the blog table updated within the past month
  - Don’t need to fetch the other columns
Data-Centric Consistency Models

• Consistency model (aka *semantics or constraints*)
  • Contract between processes and the data store
    • If processes obey certain rules, data store will work correctly
  • *All models attempt to return the results of the last write for a read operation*
    • Differ in how “last” write is determined/defined
A distributed storage system can achieve at most two of C, A, and P.

When partition-tolerance is important, you have to choose between consistency and availability.

**CAP**

**PACECLC:**

```java
if(partition) {
    choose A or C
} else {
    choose latency or consistency
}
```

- **Consistency**
  - HBase, HyperTable, BigTable, Spanner

- **Availability**
  - Cassandra, RIAK, Dynamo, Voldemort

- **Partition-tolerance**
  - RDBMSs (non-replicated)
Consistency Spectrum

- **Faster reads and writes**
- **More consistency**
- **Strong (e.g., Sequential)**

**Eventual Consistency**
- If writes to a key stop, all replicas of key will converge
- Originally from Amazon’s Dynamo and LinkedIn’s Voldemort systems

**Strong Consistency**
- See all previous writes
- ACID properties
### Some Read Consistency Guarantees

<table>
<thead>
<tr>
<th>Consistency Type</th>
<th>Description</th>
<th>Consistency</th>
<th>Performance</th>
<th>Availability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strong Consistency</td>
<td>See all previous writes.</td>
<td>A</td>
<td>D</td>
<td>F</td>
</tr>
<tr>
<td>Eventual Consistency</td>
<td>See subset of previous writes.</td>
<td>D</td>
<td>A</td>
<td>A</td>
</tr>
<tr>
<td>Consistent Prefix</td>
<td>See initial sequence of writes.</td>
<td>C</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>Bounded Staleness</td>
<td>See all “old” writes.</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>Monotonic Reads</td>
<td>See increasing subset of writes.</td>
<td>C</td>
<td>B</td>
<td>B</td>
</tr>
<tr>
<td>Read My Writes</td>
<td>See all writes performed by reader.</td>
<td>C</td>
<td>C</td>
<td>C</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Strong**
  - Prefix
  - Bounded
  - Monotonic
  - RMW
- **Eventual**

**Metric:**

\[ \text{metric} = \text{set of allowable read results} \]
Strict/Strong Consistency

• “See all previous writes”
• Any read always returns the result of the most recent write
  • Implicitly assumes the presence of a global clock
  • A write is immediately visible to all processes
    • Difficult to achieve in real systems as network delays can be variable
Sequential Consistency

- weaker than strict consistency
  - All operations are executed in some sequential order
  - each process issues operations in program order
    - Any valid interleaving is allowed
    - All agree on the same interleaving
    - Each process preserves its program order

Why is this weaker than strict/strong?

Nothing is said about “most recent write”
Linearizability

• Assumes sequential consistency and
  • If $TS(x) < TS(y)$ then $OP(x)$ should precede $OP(y)$ in the sequence
  • Stronger than sequential consistency
  • Difference between linearizability and serializability?
    • granularity: reads/writes versus transactions

• Example:

<table>
<thead>
<tr>
<th>Process P1</th>
<th>Process P2</th>
<th>Process P3</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = 1;$</td>
<td>$y = 1;$</td>
<td>$z = 1;$</td>
</tr>
<tr>
<td>print (y, z);</td>
<td>print (x, z);</td>
<td>print (x, y);</td>
</tr>
</tbody>
</table>

• Is this weaker than strict/strong?
• Still, nothing is said about “most recent write”
Linearizability/Sequential Consistency Example

- Four valid execution sequences for the processes of the previous slide. The vertical axis is time.

\[
\begin{align*}
&\text{(a)} \quad x = 1; \\
&\text{print } ((y, z); \\
&y = 1; \\
&\text{print } (x, z); \\
&z = 1; \\
&\text{print } (x, y); \\
&\text{Prints: } 001011 \\
&\text{Signature: } 001011
\end{align*}
\]

\[
\begin{align*}
&\text{(b)} \quad x = 1; \\
&y = 1; \\
&\text{print } (x,z); \\
&z = 1; \\
&\text{print } (x, y); \\
&\text{Prints: } 101011 \\
&\text{Signature: } 101011
\end{align*}
\]

\[
\begin{align*}
&\text{(c)} \quad y = 1; \\
&z = 1; \\
&\text{print } (x, y); \\
&x = 1; \\
&\text{print } (y, z); \\
&\text{Prints: } 010111 \\
&\text{Signature: } 110101
\end{align*}
\]

\[
\begin{align*}
&\text{(d)} \quad y = 1; \\
&x = 1; \\
&\text{print } (x, z); \\
&z = 1; \\
&\text{print } (x, y); \\
&\text{Prints: } 111111 \\
&\text{Signature: } 111111
\end{align*}
\]
Causal consistency

• Causally related writes seen by all processes in the same order.
  • Causally?
  • Concurrent writes may be seen in different orders on different machines

Causal:
If a write produces a value that causes another write, they are causally related

\[ X = 1 \]
\[ \text{if}(X > 0) \{ \]
\[ \quad Y = 1 \]
\[ \} \]

Causal consistency \(\rightarrow\) all see \(X=1, Y=1\) in the same order

Not permitted

Permitted
Other models

• FIFO consistency
  • Writes from a process seen by others in same order.
  • Writes from different processes may be seen in different order (even if causally related)
  • Relaxes causal consistency
  • Simple implementation:
    • tag each write by (Proc ID, seq #)

• Even FIFO consistency may be too strong
  • Requires all writes from process seen in order

• Assume use of critical sections for updates
  • Send final result of critical section everywhere
  • Do not worry about propagating intermediate results
    • Assume presence of synchronization primitives to define semantics

• Weak consistency
  • Accesses to sync var associated with a data store → sequentially consistent
  • No operation on a synchronization variable is allowed to be performed until all previous writes have been completed everywhere
  • No read or write operation on data items are allowed to be performed until all previous operations to synchronization variables have been performed.

• Entry and release consistency
  • Assume shared data are made consistent at entry or exit points of critical sections
## Summary of Data-centric Consistency Models

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Strict</td>
<td>Absolute time ordering of all shared accesses matters.</td>
</tr>
<tr>
<td>Linearizability</td>
<td>All processes must see all shared accesses in the same order. Accesses are furthermore ordered according to a (nonunique) global timestamp</td>
</tr>
<tr>
<td>Sequential</td>
<td>All processes see all shared accesses in the same order. Accesses are not ordered in time</td>
</tr>
<tr>
<td>Causal</td>
<td>All processes see causally-related shared accesses in the same order.</td>
</tr>
<tr>
<td>FIFO</td>
<td>All processes see writes from each other in the order they were used. Writes from different processes may not always be seen in that order</td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>Consistency</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Weak</td>
<td>Shared data can be counted on to be consistent only after a synchronization is done</td>
</tr>
<tr>
<td>Release</td>
<td>Shared data are made consistent when a critical region is exited</td>
</tr>
<tr>
<td>Entry</td>
<td>Shared data pertaining to a critical region are made consistent when a critical region is entered.</td>
</tr>
</tbody>
</table>

(b)