CS-562 Advanced Topics in Databases

Haridimos Kondylakis

Computer Science Department, University of Crete
What is MongoDB?

- Developed by 10gen: “humongous” DB
  - Founded in 2007
- A document-oriented NoSQL database
- Hash-based, **schema-less database**
  - No Data Definition Language
  - Can store hashes with any keys and values that you choose
    - Keys are a basic data type but in reality stored as strings
    - Document Identifiers (_id) will be created for each document, field name reserved by system
  - Uses BSON format
    - Based on JSON – B stands for Binary
- Supports APIs (drivers) in many computer languages
  - JavaScript, Python, Ruby, Perl, Java, Java Scala, C#, C++, Haskell, Erlang
Focus on Consistency and Partition tolerance

- **Consistency**
  - all replicas contain the same version of the data

- **Availability**
  - system remains operational on failing nodes

- **Partition tolerance**
  - multiple entry points
  - system remains operational on system split

**CAP Theorem**: satisfying all three at the same time is impossible
Why use MongoDB?

✓ MongoDB Features
  • Document-Oriented storage
  • Full Index Support
  • Replication & High Availability
  • Auto-Sharding
  • Ad-hoc Querying
  • Fast In-Place Updates
  • Map/Reduce functionality

✓ Application need
  • Simple queries
  • Functionality provided applicable to most web applications
  • Easy and fast integration of data
  • Not well suited for heavy and complex transactions systems
MongoDB: Hierarchical Objects

- A MongoDB instance may have zero or more ‘databases’.
- A database may have zero or more ‘collections’.
- A collection may have zero or more ‘documents’.
- A document may have one or more ‘fields’.
- MongoDB ‘Indexes’ function much like their RDBMS counterparts.
# RDB Concepts to NO SQL

<table>
<thead>
<tr>
<th>RDBMS</th>
<th>MongoDB</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Database</strong></td>
<td>Database</td>
</tr>
<tr>
<td><strong>Table, View</strong></td>
<td><strong>Collection</strong></td>
</tr>
<tr>
<td><strong>Row</strong></td>
<td><strong>Document (BSON)</strong></td>
</tr>
<tr>
<td><strong>Column</strong></td>
<td><strong>Field</strong></td>
</tr>
<tr>
<td><strong>Index</strong></td>
<td><strong>Index</strong></td>
</tr>
<tr>
<td><strong>Join</strong></td>
<td><strong>Embedded Document</strong></td>
</tr>
<tr>
<td><strong>Foreign Key</strong></td>
<td><strong>Reference</strong></td>
</tr>
<tr>
<td><strong>Partition</strong></td>
<td><strong>Shard</strong></td>
</tr>
</tbody>
</table>
BSON

• “Binary JSON”
• Binary-encoded serialization of JSON-like docs
• Also allows “referencing”
• Embedded structure reduces need for joins

Goals
• Lightweight
• Traversable
• Efficient (decoding and encoding)

{"hello": "world"} →  \\x15\x00\x00\x00
  \x02
  hello\x00
  \x05\x00\x00\x00world\x00
  \x00
  // total document size
  // 0x02 = type String
  // field name
  // field value
  // 0x00 = type EOF ('end of object')
The _id Field

• By default, each document contains an _id field. This field has a number of special characteristics:
  • Value serves as primary key for collection.
  • Value is unique, immutable, and may be any non-array type.
  • Default data type is ObjectId, which is “small, likely unique, fast to generate, and ordered.”

```json
{ "_id" : "37010",
  "city" : "ADAMS",
  "pop" : 2660,
  "state" : "TN" }
```

http://docs.mongodb.org/manual/reference/bson-types/
BSON Example

```json
{ 
"_id" : "37010",
"city" : "ADAMS",
"pop" : 2660,
"state" : "TN",
"councilman" : { 
name: "John Smith",
address: "13 Scenic Way"
}

}

{ 
"_id" : "1",
"first name" : "Hassan",
"last name" : "Mir",
"department" : 20

}

{ 
"_id" : "1",
"first name" : "Bill",
"last name" : "Gates"

}
```
CRUD

• Create
  • db.collection.insert( <document> )
  • db.collection.save( <document> )
  • db.collection.update( <query>, <update>, { upsert: true } )

• Read
  • db.collection.find( <query>, <projection> )
  • db.collection.findOne( <query>, <projection> )

• Update
  • db.collection.update( <query>, <update>, <options> )

• Delete
  • db.collection.remove( <query>, <justOne> )
CRUD example

```javascript
> db.user.insert({
  first: "John",
  last : "Doe",
  age: 39
})

> db.user.find ()
{
   "_id" : ObjectId("51..."),
   "first" : "John",
   "last" : "Doe",
   "age" : 39
}

> db.user.update(
   {
     "_id" : ObjectId("51...")},
   {
     $set: {
       age: 40,
       salary: 7000
     }
   }
)

> db.user.remove({
   "first": /^J/ 
})
```
Replication of data

- Ensures redundancy, backup, and automatic failover
  - Recovery manager in the RDMS
  - Replication through groups of servers known as replica sets
    - Primary set – set of servers that client tasks direct updates to
    - Secondary set – set of servers used for duplication of data
- If the primary set fails the secondary sets ‘vote’ to elect the new primary set
Replica Sets

- Redundancy and Failover
- Zero downtime for upgrades and maintenance

- Master-slave replication
  - Strong Consistency
  - Delayed Consistency

- Geospatial features
Consistency of data

- All read operations issued to the primary of a replica set are consistent with the last write operation
  - Reads to a primary have **strict consistency**
    - Reads reflect the latest changes to the data
  - Reads to a secondary have **eventual consistency**
    - Updates propagate gradually
  - If clients permit reads from secondary sets – then client may read a previous state of the database
- Failure occurs before the secondary nodes are updated
  - System identifies when a rollback needs to occur
  - Users are responsible for manually applying rollback changes
Sharding

- Partition your data
- Scale write throughput
- Increase capacity
- Auto-balancing

<table>
<thead>
<tr>
<th>id</th>
<th>company</th>
<th>customer</th>
<th>article</th>
<th>currency</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>4250250</td>
<td>020</td>
<td>073000</td>
<td>5994537812</td>
<td>00</td>
<td>142,50</td>
</tr>
<tr>
<td>4250251</td>
<td>020</td>
<td>073000</td>
<td>5994537852</td>
<td>00</td>
<td>141,12</td>
</tr>
<tr>
<td>4250252</td>
<td>020</td>
<td>073000</td>
<td>5994537854</td>
<td>00</td>
<td>105,99</td>
</tr>
<tr>
<td>4250253</td>
<td>020</td>
<td>073000</td>
<td>5994537856</td>
<td>00</td>
<td>109,52</td>
</tr>
<tr>
<td>4250254</td>
<td>020</td>
<td>073000</td>
<td>5994537862</td>
<td>00</td>
<td>131,49</td>
</tr>
<tr>
<td>4250255</td>
<td>020</td>
<td>073000</td>
<td>5994567308</td>
<td>00</td>
<td>29,86</td>
</tr>
<tr>
<td>4250256</td>
<td>020</td>
<td>073000</td>
<td>5994567422</td>
<td>00</td>
<td>57,13</td>
</tr>
<tr>
<td>4250257</td>
<td>020</td>
<td>073000</td>
<td>5994567428</td>
<td>00</td>
<td>68,59</td>
</tr>
<tr>
<td>4250258</td>
<td>020</td>
<td>073000</td>
<td>5994605089</td>
<td>00</td>
<td>51,09</td>
</tr>
<tr>
<td>4250259</td>
<td>020</td>
<td>073000</td>
<td>5994607975</td>
<td>00</td>
<td>93,93</td>
</tr>
<tr>
<td>4250260</td>
<td>020</td>
<td>073000</td>
<td>5994701005</td>
<td>00</td>
<td>74,22</td>
</tr>
</tbody>
</table>
Choices made for Design of MongoDB

- Scale horizontally over commodity hardware
  - Lots of relatively inexpensive servers
- Keep the functionality that works well in RDBMSs
  - Ad hoc queries
  - Fully featured indexes
  - Secondary indexes
Column store
Row Store and Column Store

- In row store data are stored in the disk tuple by tuple.
- Where in column store data are stored in the disk column by column.
Row Store and Column Store

• Most of the queries does not process all the attributes of a particular relation.

• For example the query
  ✓ Select c.name and c.address
  ✓ From CUSTOMES as c
  ✓ Where c.region=Mumbai;

• Only process three attributes of the relation CUSTOMER. But the customer relation can have more than three attributes.

• Column-stores are more I/O efficient for read-only queries as they read, only those attributes which are accessed by a query.
Row Store and Column Store

<table>
<thead>
<tr>
<th>Row Store</th>
<th>Column Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>(+) Easy to add/modify a record</td>
<td>(+) Only need to read in relevant data</td>
</tr>
<tr>
<td>(-) Might read in unnecessary data</td>
<td>(-) Tuple writes require multiple accesses</td>
</tr>
</tbody>
</table>

- So column stores are suitable for **read-mostly, read-intensive, large data repositories**
Why Column Stores?

- Can be significantly faster than row stores for some applications
  - Fetch only required columns for a query
  - Better cache effects
  - Better compression (similar attribute values within a column)
- But can be slower for other applications
  - OLTP with many row inserts, ..
- Long war between the column store and row store camps :-}
Column Stores - Data Model

• Standard relational logical data model
  • EMP(name, age, salary, dept)
  • DEPT(dname, floor)

• Table – collection of projections

• Projection – set of columns

• Horizontally partitioned into segments with segment identifier
Column Stores - Data Model

• To answer queries, projections are joined using Storage keys and join indexes

• Storage Keys:
  • Within a segment, every data value of every column is associated with a unique Skey
  • Values from different columns with matching Skey belong to the same logical row
• Join Indexes
  • T1 and T2 are projections on T
  • M segments in T1 and N segments in T2
  • Join Index from T1 to T2 is a table of the form:
    • (s: Segment ID in T2, k: Storage key in Segment s)
    • Each row in join index matches corresponding row in T1
  • Join indexes are built such that T could be efficiently reconstructed from T1 and T2
Column Stores – Data Model

• Construct EMP(name, age, salary) from EMP1 and EMP3 using join index on EMP3
Compression

• Trades I/O for CPU
  • Increased column-store opportunities:
    • Higher data value locality in column stores
    • Data compression techniques such as run length encoding far more useful

• Schemes
  • Null Suppression
  • Dictionary encoding
  • Run Length encoding
  • Bit-Vector encoding
  • Heavyweight schemes
Query Execution - Operators

- **Select**: Same as relational algebra, but produces a bit string
- **Project**: Same as relational algebra
- **Join**: Joins projections according to predicates

- **Aggregation**: SQL like aggregates
- **Sort**: Sort all columns of a projection
Query Execution - Operators

- **Decompress**: Converts compressed column to uncompressed representation
- **Mask**\((\text{Bitstring } B, \text{Projection } Cs) \Rightarrow \text{emit only those values whose corresponding bits are } 1\)
- **Concat**: Combines one or more projections sorted in the same order into a single projection
- **Permute**: Permutes a projection according to the ordering defined by a join index
- **Bitstring operators**: \(\text{Band} – \text{Bitwise AND, Bor} – \text{Bitwise OR, Bnot} – \text{complement}\)
Row Store Vs Column Store

- the difference in storage layout leads to that one can obtain the performance benefits of a column-store using a row-store by making some changes to the physical structure of the row store.

- This changes can be
  - Vertically partitioning
  - Using index-only plans
  - Using materialized views
Vertical Partitioning

• **Process:**
  • Full Vertical partitioning of each relation
    • Each column = 1 Physical table
    • This can be achieved by adding integer position column to every table
    • Adding integer position is better than adding primary key
  • Join on Position for multi column fetch

• **Problems:**
  • “Position” - Space and disk bandwidth
  • Header for every tuple – further space wastage
    • e.g. 24 byte overhead in PostgreSQL
Vertical Partitioning: Example

[Diagram showing vertical partitioning of data]
Index-only plans

• Process:
  • Add B+Tree index for every Table.column
  • Plans never access the actual tuples on disk
  • Headers are not stored, so per tuple overhead is less

• Problem:
  • Separate indices may require full index scan, which is slower
  • Eg: SELECT AVG(salary)
        FROM emp
        WHERE age > 40
  • Composite index with (age, salary) key helps.
Index-only plans: Example
Materialized Views

• **Process:**
  • Create ‘optimal' set of MVs for given query workload
  • **Objective:**
    • Provide just the required data
    • Avoid overheads
    • Performs better

• **Expected to perform better than other two approach**

• **Problems:**
  • Practical only in limited situation
  • Require knowledge of query workloads in advance
Materialized Views: Example

- Select F.custID
  from Facts as F
  where F.price > 20
Optimizing Column oriented Execution

• Different optimization for column oriented database
  • Compression
  • Late Materialization
  • Block Iteration
Compression

• Low information entropy (high data value locality) leads to High compression ratio

• Advantage
  • Disk Space is saved
  • Less I/O
  • CPU cost decrease if we can perform operation without decompressing

• Light weight compression schemes do better
Compression

• If data is sorted on one column that column will be super-compressible in row store

• eg. Run length encoding
Late Materialization

• Most query results entity-at-a-time not column-at-a-time
• So at some point of time multiple column must be combined
• One simple approach is to join the columns relevant for a particular query
• But further performance can be improve using late-materialization

‣ Idea: Delay Tuple Construction
‣ Might avoid constructing it altogether
‣ Intermediate position lists might need to be constructed
‣ Eg: SELECT R.a FROM R WHERE R.c = 5 AND R.b = 10
  ‣ Output of each predicate is a bit string
  ‣ Perform Bitwise AND
  ‣ Use final position list to extract R.a
Late Materialization

• Advantages
  • Unnecessary construction of tuple is avoided
  • Direct operation on compressed data
  • Cache performance is improved
Block Iteration

- Operators operate on blocks of tuples at once
- Iterate over blocks rather than tuples
- Like batch processing
- If column is fixed width, it can be operated as an array
- Minimizes per-tuple overhead
- Exploits potential for parallelism
- Can be applied even in Row stores – IBM DB2 implements it
Graph databases
What is a Graph Database?

• A database with an explicit graph structure
• Each node knows its adjacent nodes
• As the number of nodes increases, the cost of a local step (or hop) remains the same
• Plus an Index for lookups
• Express Queries as Traversals. Fast deep traversal instead of slow SQL queries that span many table joins.
• Very natural to express graph related problem with traversals (recommendation engine, find shortest path etc..)
• Seamless integration with various existing programming languages.
• ACID Transaction with rollbacks support.
• Distinguish between “Database for graph as object”!
Social Network “path exists” Performance

• Experiment:
  • ~1k persons
  • Average 50 friends per person
  • `pathExists(a, b)` limited to depth 4

<table>
<thead>
<tr>
<th></th>
<th># persons</th>
<th>query time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relational</td>
<td>1000</td>
<td>2000ms</td>
</tr>
<tr>
<td>database</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Neo4j</td>
<td>1000</td>
<td>2ms</td>
</tr>
<tr>
<td>Neo4j</td>
<td>1000000</td>
<td>2ms</td>
</tr>
</tbody>
</table>
Compared to Relational Databases

Optimized for aggregation

Optimized for connections
Compared to Key Value Stores

Optimized for simple look-ups

Optimized for traversing connected data
Compared to Document Stores

Optimized for “trees” of data

Optimized for seeing the forest and the trees, and the branches, and the trunks
Property Graph
What is Neo4j?

- A Graph Database + Lucene Index
- Property Graph
- Full ACID (atomicity, consistency, isolation, durability)
- High Availability (with Enterprise Edition)
- 32 Billion Nodes, 32 Billion Relationships, 64 Billion Properties
- Embedded Server
- REST API
Good For

- Highly connected data (social networks)
- Recommendations (e-commerce)
- Path Finding (how do I know you?)

- A* (Least Cost path)
- Data First Schema (bottom-up, but you still need to design)
Summary: noSQL Common Advantages

- Cheap, easy to implement (open source)
- Data are replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
- Easy to distribute
- Don't require a schema
- Can scale up and down
- Relax the data consistency requirement (CAP)
Summary: What are we giving up?

- joins
- group by
- order by
- ACID transactions (none are strict ACID!)
- SQL as a sometimes frustrating but still powerful query language
- easy integration with other applications that support SQL
**HOW TO WRITE A CV**

**Percentage of Matching Job Postings**

**Leverage the NoSQL boom**

**DO YOU HAVE ANY EXPERTISE IN SQL?**

**NO**

**DOESN’T MATTER. WRITE: "EXPERT IN NO SQL"**