# Big Data Engineering and Architecture

Topic 5: Storage and Serialization

# Data Modeling

- Generically: representation of data in terms of the next-lower layer
  - Business App → APIs, objects/data structures
  - Storing app data structures → DBs, files, etc.
  - DB storage → memory, disk, network for usage
  - Disk/memory → hardware representation (SSD, HDD, etc.)
- Each layer hides the implementation below it
- MUST choose the “right” model
  - Csci 4707/5707: Relational data
  - This class: NoSQL data models and distributed data

# Impedance Mismatch

- The disconnect between the models of two layers
- Business application uses objects, RDBMS uses tables
  - Translation is required from objects → tables
- Code must be written to translate
- This also occurs when a storage technology isn’t ideal
  - E.g.: Storing relationship data (graph-like) in a key-value store (key lookup)
  - E.g.: Storing nested JSON in RDBMS or Column Family
- Try this in your project, it will earn you kudos

# Locality

- RDBMSs store data in tables
  - Tables are not necessarily near each other on disk
  - Related data for an entity normalized
  - Joins are required to retrieve related data ➔ Multiple seeks
  - What effect does this have on analytics?
- Denormalization improves locality
  - Data for a record is pre-joined
  - All related data stored in same row
  - One seek to retrieve related data
- NoSQL
  - Nested JSON
  - Column families
- What are the hardware impacts (positive/negative) of locality?
- What's the proper level of locality / denormalization?

# Joins in NoSQL

- Joins are often not supported
- Joins are also not often required if proper modeling is used
- Inevitably, joins will be requested. Why?
  - If your system **is not** designed properly: users need usable data
  - If your system **is** designed properly: it’s providing value and curiosity ensues
  - Data use cases evolve over time (design can be futile)
- How to handle need for joins?
  - Emulate a join in application code (application layer)
  - Evolve data schema

# Joins in Application Layer: Document DBs

```
{
    "id": 1,
    "name": "Peter Griffin",
    "spouse_ids": [2],
    "children_ids": [3,4,5]
},
{
    "id": 2,
    "name": "Lois Griffin",
    "spouse_ids": [1],
    "children_ids": [3,4,5]
},
{
    "id": 3,
    "name": "Meg Griffin"
},
{
    "id": 4,
    "name": "Chris Griffin"
},
{
    "id": 5,
    "name": "Stewie Griffin"
}
```

1. Describe the locality of each
2. Will a join be required?
    1. What types of BQs will require joins?
    2. Which BQs don’t require joins?
3. What are the impacts of updates to
    1. Names of embedded entities?
    2. Adding nickname to each person?

```
{
    "id": 1,
    "name": "Peter Griffin",
    "spouses": [
        {
            "id": 2,
            "name": "Lois Griffin",
            // should children go here?
        }
    ],
    "children": [
        {
            "id": 3,
            "name": "Meg Griffin"
        },
        {
            "id": 4,
            "name": "Chris Griffin"
        },
        {
            "id": 5,
            "name": "Stewie Griffin"
        }
    ]
}
```

# Joins in Application Layer: Key/Value DBs

```
person_1_name        => "Peter Griffin"
person_1_spouseids   => "2"
person_1_childrenids => "3,4,5"
person_2_name        => "Lois Griffin"
person_2_spouseids   => "1"
person_2_childrenids => "3,4,5"
person_3_name        => "Meg Griffin"
person_4_name        => "Chris Griffin"
person_5_name        => "Stewie Griffin"
```

What are some models to reduce need for application joins?

# So, Which is Better to Use?

- RDBMS
- Document Database
- Key/Value Database
- Column Family Database
- Graph Database

# Schema

The “structure” or definition of data elements

# Schema Concepts

- NoSQL doesn't always **enforce** schema
- Some schema is assumed, and should be designed/agreed to
  - Document/describe as part of your data model design
- Schemaless
  - No guaranteed/enforced structure of keys/values
- Schema-on-read
  - Schema of the data is interpreted when read
- Schema-on-write
  - Schema is enforced upon write (E.g.: RDBMS)

# Schema Evolution

- Schema change on schema-on-write DB
  - ALTER statements
  - Impacts:
    - Careful planning and migration
    - Slow, requires downtime
- Schema change on schema-on-read DB
  - Can add/remove/modify attributes at will
  - What impacts will that have?
- More on evolution later…

# Storage and Retrieval

# Storage and the Business Question

The Business Question (or application) is important!

1. Choosing the appropriate storage engine
2. Tuning the storage engine
   1. Tune the model and storage implementation
   2. Need to know a little about how the storage engine works

# Data Structures

- Files
  - Appending is super efficient
  - Databases often use a log (append-only)
    - E.g.: Oracle change data capture (CDC)
- Index
  - Maintenance of each index incurs overhead (additional writes)
  - Trade-off: Faster reads, slower writes
  - Choose your indexes wisely – how do you know what to choose?
- Hash, SSTable/LSM-Tree, B-Tree

# Appending to Files

- Records can be read or appended, not updated
- How to remove previous writes for a record?
  - Compaction of segment files
  - Write a **new** log with only latest versions. Delete the old files.
  - Sometimes you have to do this manually
- Compaction can be done in the background
- Why append-only?
  - Sequential writes are faster than random writes
    - Faster on spinning drives
    - Likely preferred on SSD
  - Easier crash recovery
  - Compaction is easy

# Hash Indexes

- Hash key to identify record location
- The good: fast, well-known, avoid storage hotspots
- The bad:
  - Must fit entire hash table in memory
  - Need to be able to handle crashes
  - Range queries are not efficient because hash(123) will not be near hash(124)

# Index/Log File Improvement

- Append-only is fast, but can it be improved?
- Requirement: sequence of key-value pairs is sorted
- New data structure: Sorted String Table (SSTable)
- Improvements on Hash Index
  - Merging is more efficient, algorithmically
  - Don't need to store **all** keys in memory
    - Reduce seeks by using known keys that are in memory
  - Group records into a **block** on disk, compressed
    - Less space on disk, less I/O bandwidth

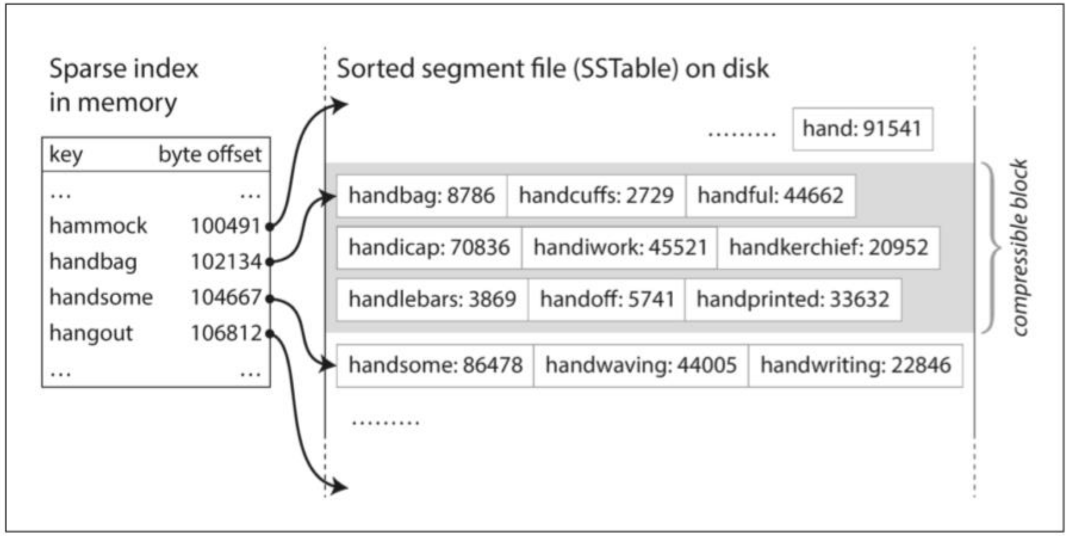

*Figure 3-5. An SSTable with an in-memory index.*

# Speed SSTable Up More!

- Easier to maintain sorted keys in memory
  1. Store writes in in-memory tree (memtable)
  2. When memtable gets large enough, write SSTable file
  3. Serving a read? Check memtable first (aka cache)
- What happens if DB crashes?
  - Memtable is gone! (records not in a SSTable)
  - Use a log (write-ahead log)
  - When memtable is persisted to disk, delete WAL
- Also known as Log-Structured Merge Tree (LSM-Tree)
- Bloom filters
- LSM-Tree supports VERY HIGH write throughput

# B-Trees

- Most common type of index, especially in RDBMSs
- Breaks database into fixed-size blocks (pages)
- Each page is addressed
- Need to keep it balanced
  - n keys depth: O(log n)
- Other terms to know
  - Branching factor
  - Write-ahead log (WAL)

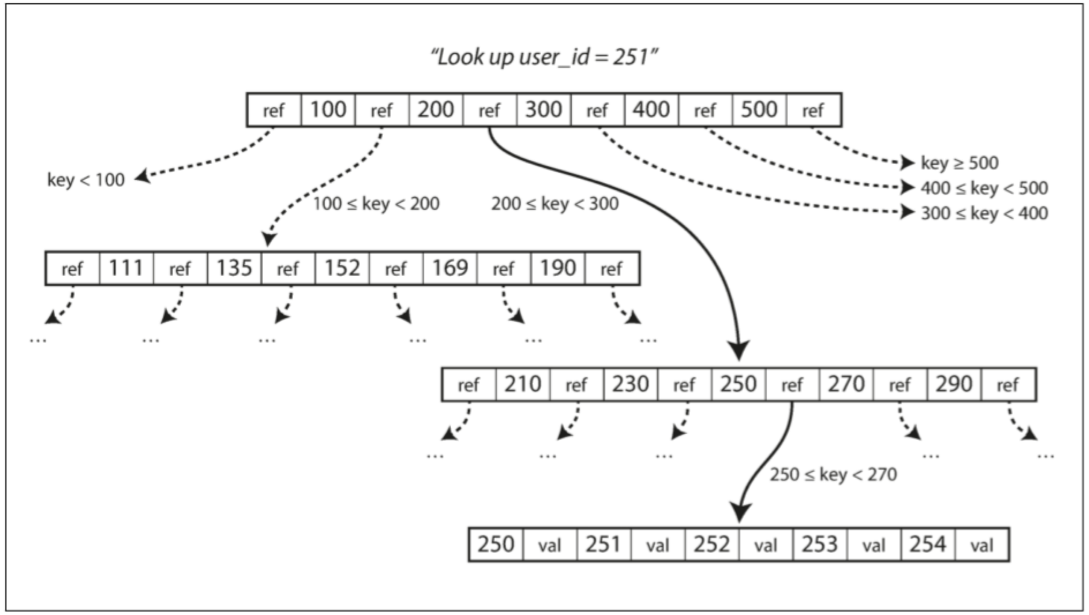


*Figure 3-6. Looking up a key using a B-tree index.*

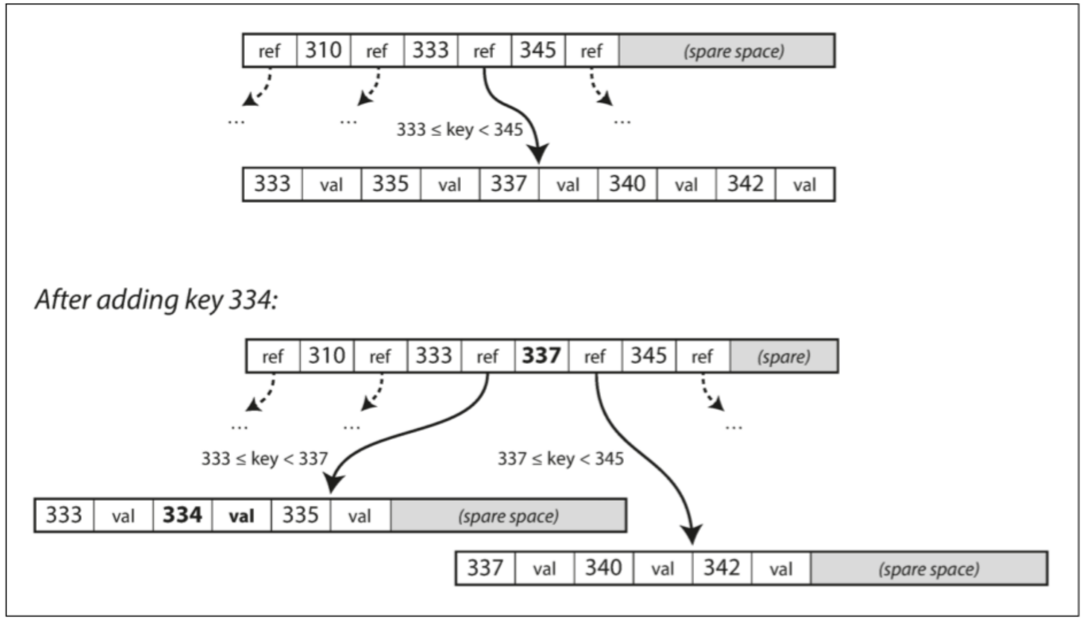


*Figure 3-7. Growing a B-tree by splitting a page.*

# Comparing LSM-Trees and B-Trees

- Faster writes: LSM-Trees
- Faster reads: B-Trees
- Multiple writes operations per write request (write amplification)
  - LSM-Trees: Segments, compaction
  - B-Trees: Page updates
  - Could be problematic in SSDs because they wear down
  - The more writes to disk that are needed, the fewer writes per second that can be handled
- B-Trees have higher write amplification
- Compaction uses valuable resources (LSM-Trees)
  - Configure compaction carefully
- B-Trees: Each key exists in exactly one place in the index
  - Good for transactions

# Other Indexes

- Secondary index
- Clustered index
- Covering index
- Concatenated index (multi-column index)

# Storage Orientation

# Types of Storage Orientation

To choose the right data storage technique, you MUST know if it is using row-oriented or column-oriented storage.

- **Row-oriented:** All values for a row are stored together
  - When you often query most/all of the columns
- **Column-oriented:** All values for a column are stored together
  - When you often query a subset of columns (analytics)

# Column-Oriented Example

fact_sales table

| date_key | product_sk | store_sk | promotion_sk | customer_sk | quantity | net_price | discount_price |
|---|---|---|---|---|---|---|---|
| 140102 | 69 | 4 | NULL | NULL | 1 | 13.99 | 13.99 |
| 140102 | 69 | 5 | 19 | NULL | 3 | 14.99 | 9.99 |
| 140102 | 69 | 5 | NULL | 191 | 1 | 14.99 | 14.99 |
| 140102 | 74 | 3 | 23 | 202 | 5 | 0.99 | 0.89 |
| 140103 | 31 | 2 | NULL | NULL | 1 | 2.49 | 2.49 |
| 140103 | 31 | 3 | NULL | NULL | 3 | 14.99 | 9.99 |
| 140103 | 31 | 3 | 21 | 123 | 1 | 49.99 | 39.99 |
| 140103 | 31 | 8 | NULL | 233 | 1 | 0.99 | 0.99 |

Columnar storage layout:

date_key file contents: 140102, 140102, 140102, 140102, 140103, 140103, 140103, 140103
product_sk file contents: 69, 69, 69, 74, 31, 31, 31, 31
store_sk file contents: 4, 5, 5, 3, 2, 3, 3, 8
promotion_sk file contents: NULL, 19, NULL, 23, NULL, NULL, 21, NULL
customer_sk file contents: NULL, NULL, 191, 202, NULL, NULL, 123, 233
quantity file contents: 1, 3, 1, 5, 1, 3, 1, 1
net_price file contents: 13.99, 14.99, 14.99, 0.99, 2.49, 14.99, 49.99, 0.99
discount_price file contents: 13.99, 9.99, 14.99, 0.89, 2.49, 9.99, 39.99, 0.99

*Figure 3-10. Storing relational data by column, rather than by row.*

# Why Column-Oriented?

- Better compression algorithms if all data types are the same
  - E.g.: Run-length encoding, bitmap encoding
- Read only required values (columns), not all values from disk
  - Lower volume reads from disk
  - Efficient use of CPU cycles (L1 cache?)
- Vectorized processing
- Can’t use B-Trees, use LSM-Trees