A Primer on Database Architectures

Part I: What Is A Database?

There are just three intrinsic properties in a database:

- Persistent insert
- Persistent read
- Delete

That's it!

A database is *not* required to have:

- Transactions
- SQL

What is **SQL**?

SQL is a declarative programming language for querying databases.

```sql
SELECT (name, job_title AS role, job_id) FROM employees;
```

```sql
INSERT INTO employees ("Aleksey Bilogur", "Glue Person", 42);
```

```sql
SELECT employees.id, jobs.job_id
    FROM employees
    INNER JOIN jobs ON employee.job_id=jobs.job_id
```

SQL is a specification, which is implemented to various degrees by various different database vendors.

It's supposed to be *portable* and *read like plain English*.

What is a **transaction**?

```
db = create_database_connection()
transaction.begin()
db.exec("INSERT INTO table ('a', 'b', 'c')")
db.exec("INSERT INTO table ('d', 'e', 'f')")
try:
    db.commit()
except:
    db.rollback()
transaction.close()
```

Until the NoSQL movement gained traction in ~2010, all commonly used databases used SQL.

Some databases shipped without it (you would write queries using the vendor's Python/JS/etcetera API package instead).

In 2021, databases are mostly back to shipping with SQL support (including the ones that started without it!).

Part II: How Databases Store Data

Relational database storage engines use one of three algorithms:

- Append-only log
- SSTables
- B-Trees

(non-relational databases may do something else entirely)

Append-only logs are the simplest:

```
#!/bin/bash

db_set () {
    echo "$1,$2" >> database
}

db_get () {

    grep "^$1," database | sed -e "s/^$1,//" | tail -n 1
}
```

```
$ db_set 123456 '{"name":"London","attractions":["Big Ben","London Eye"]}'

$ db_set 42 '{"name":"San Francisco","attractions":["Big Ben","Golden Gate Bridge"]}'

$ cat database
123456,{"name":"London","attractions":["Big Ben","London Eye"]}
42,{"name":"San Francisco","attractions":["Big Ben","Golden Gate Bridge"]}

$ db_get 42
{"name": "San Francisco","attractions":["Golden Gate Bridge"]}
```

Writes are `O(1)` (append), good.

Reads are `O(n)` (full file scan), bad.

How do we improve this?

One improvement—ocassionally apply **compaction** to the data:

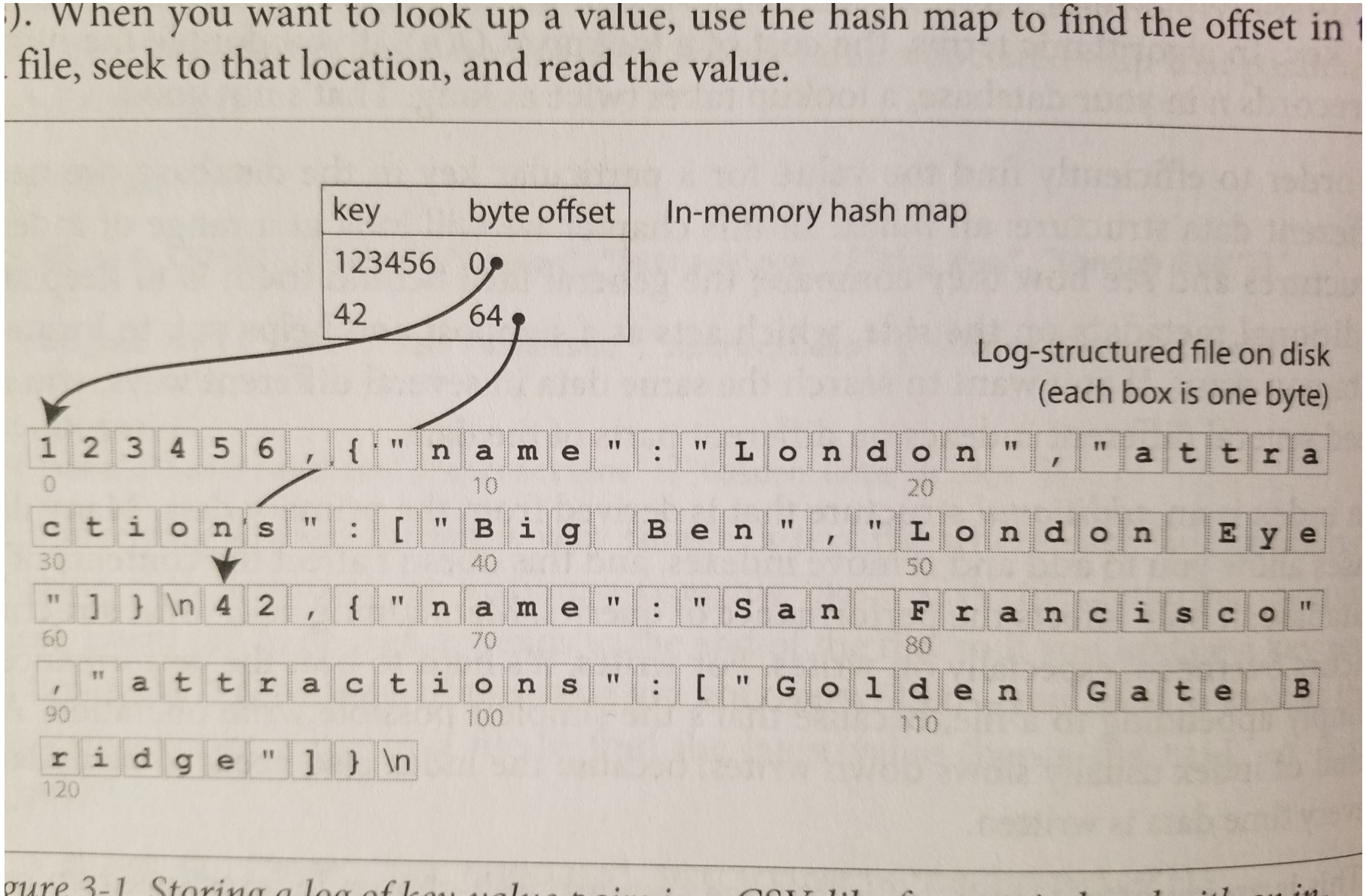
```
mew,1078,
purr,2103,
purr,2104,
mew,1079,
mew,1080,
mew,1081,

purr,2105,
yawn,511,
```

Compacted becomes:

```
mew,1081,
purr,2105,
yawn,511,
```

Which is much faster to scan!

Another—use an **index.**



This give us `O(1)` read and `O(1)` write performance.

However, range queries are still `O(n)`.

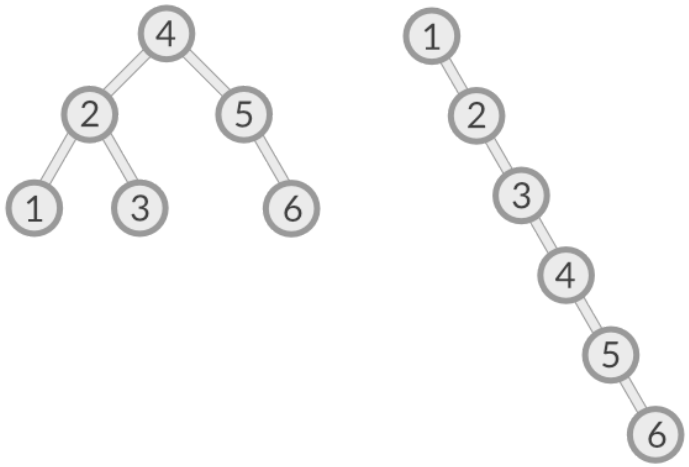
Also, the hash table must fit into memory.

Next, **SSTables**.

But first we'll need a couple of new tools.

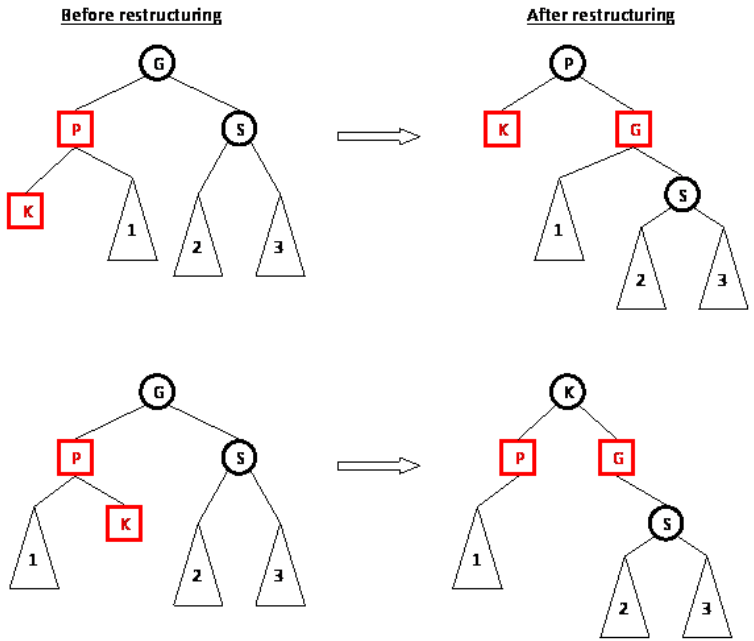
The SSLTables algorithm uses a **balanced tree algorithm**.

A tree with `n` nodes is said to be balanced if it has a height of at most `log(n)`.



There are certain algorithms that gaurantee that the tree will be balanced no matter what order of inserts, updates, and deletes you use.

These use node rotations and are typically taught in an intro to data structures class:



Key feature: balanced tree algorithms let you insert

key in any order, then iterate over them in sorted order in linear time.

SSLTables also needs **segmentation**.

Instead of using one log file, SSLTables has multiple log files ("segments").

Compaction compresses these many log segments into a single log segment.

The full SSLTables write algorithm:

1. When a write comes in, store it in an in-memory balanced tree (the "memtable").
2. When the memtable exceeds a certain size, write it to disk as an (ordered) log segment.
3. When compaction kicks off, simultaneously iterate through all of the log files in sorted, last-write-wins order. Write the "winning" record to the new log file.

The full SSLTables read algorithm:

1. When a read comes in, check the memtable for the data.
   Because the memtable is a balanced tree, it can be searched in `O(log n)` time.
2. If it's not there, check the most recent log segment.
   Because the log segment is sorted, it can be searched in `O(log n)` time.
3. If it's not there either, check the second most recent log segment.
4. Repeat you find it.

Performance considerations:

- Append-only logs offers `O(1)` writes, SSLTables offers `O(log n)` writes.
- Append-only logs offer `O(1)` reads, subject to the restriction that the hash map fits in memory. Otherwise read performance degrades to `O(n)`.
- SSLTables offers `O(log n)` writes **with no restrictions on database size**.

  This is because the log files are in sorted order! If all keys are the same size, log files can be binary searched. If keys can be different sizes, a sparse index can be used to achieve the same (amortized) time.
- Append-only logs have `O(n)` range queries (e.g. "find all words between 'cat' and 'cop'"). SSLTables have `O(log n)` range queries.

Next, **B-Trees**.

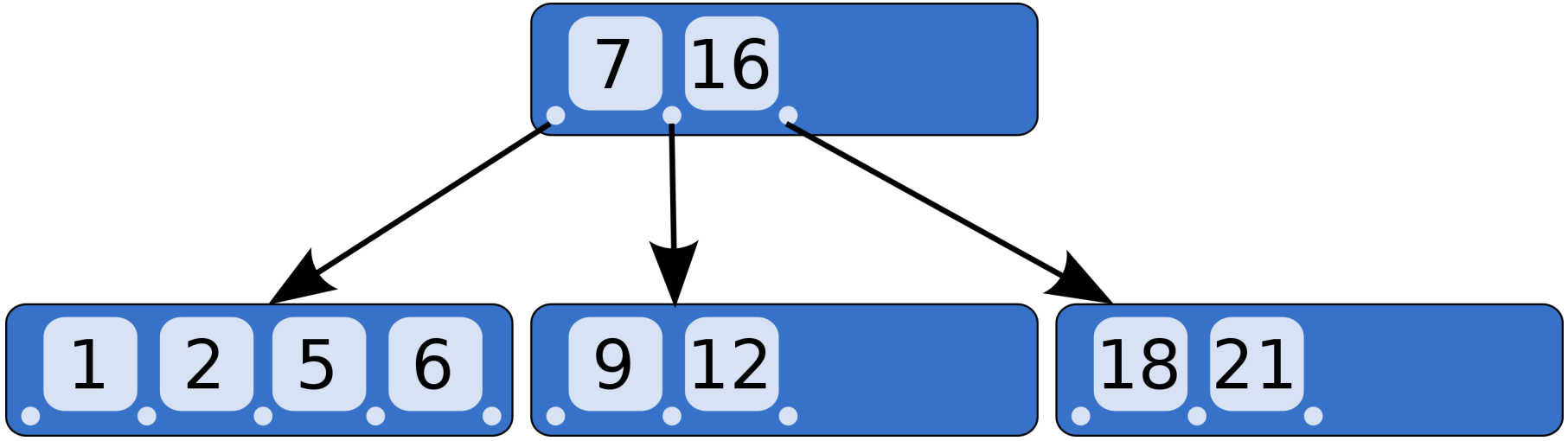
All of the "classical" databases use this algorithm.

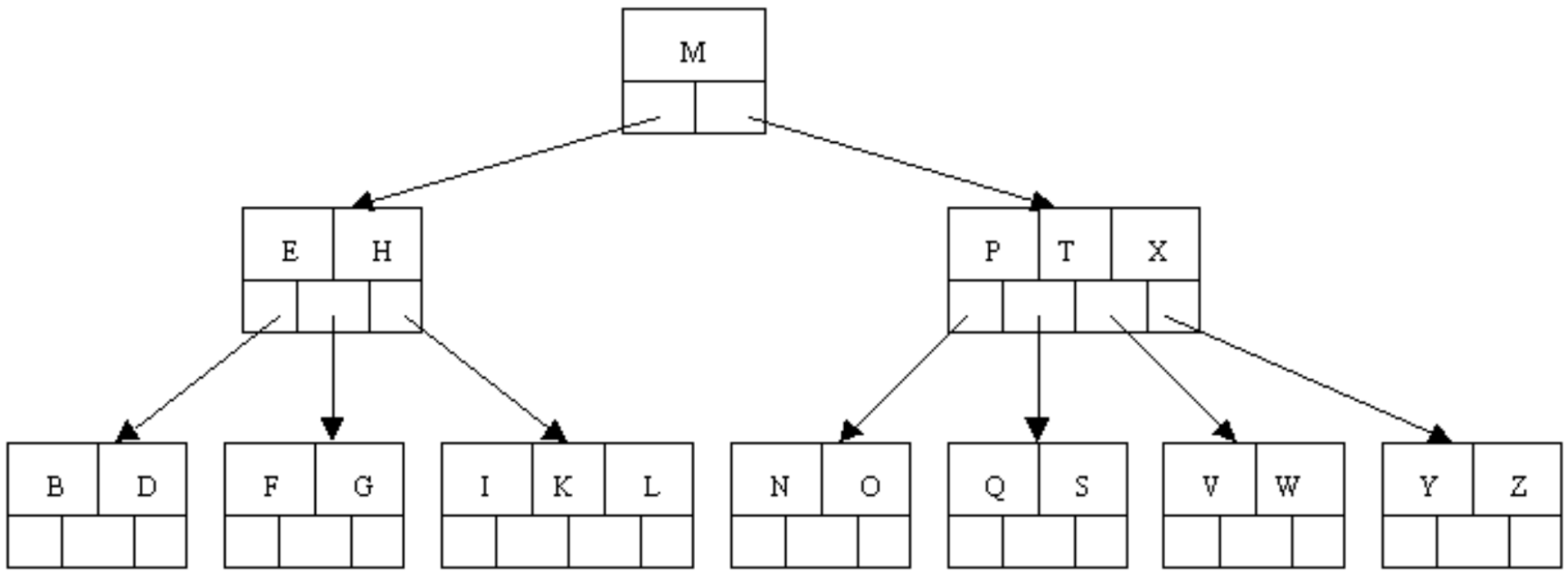
A B-tree is a balanced tree algorithm that is optimized for storage on disk.

New idea: **pages.**

Page sizes among architectures[17]

Architecture	Smallest page size	Larger page sizes
32-bit x86[18]	4 KiB	4 MiB in PSE mode, 2 MiB in PAE mode[19]
x86-64[18]	4 KiB	2 MiB, 1 GiB (only when the CPU has `PDPE1GB` flag)
IA-64 (Itanium)[20]	4 KiB	8 KiB, 64 KiB, 256 KiB, 1 MiB, 4 MiB, 16 MiB, 256 MiB[19]
Power ISA[21]	4 KiB	64 KiB, 16 MiB, 16 GiB
SPARC v8 with SPARC Reference MMU[22]	4 KiB	256 KiB, 16 MiB
UltraSPARC Architecture 2007[23]	8 KiB	64 KiB, 512 KiB (optional), 4 MiB, 32 MiB (optional), 256 MiB (optional), 2 GiB (optional), 16 GiB (optional)
ARMv7[24]	4 KiB	64 KiB, 1 MiB ("section"), 16 MiB ("supersection") (defined by a particular implementation)


7
16
1
2
5
6
9
12
18
21


M
E H
P T X
B D
F G
I K L
N O
Q S
V W
Y Z

For safety, you also need a **write-ahead log**.

Part III: Data Models In Use Today

relational

column-oriented stores

in-memory kv stores

actual usage

document stores

wide-column stores

graph

RELATIONAL DATABASES

- SQLite, PostGres, MySQL, MariaDB, etcetera.
- Relational schema, SQL query language.
- Optimized for online transaction processing.
- But can do almost anything these days.
- Want to use something else instead?

- A simple file-based DB (no managed DB process).
- No concurrency—all requests are serial.
- Transactions.
- Limited SQL support.
- Open source.

- The most popular full-featured relational DB.
- Concurrency, transactions.
- Broad SQL support.
- Open source.
- Great plugin ecosystem (PostGIS etcetera).

- The second (?) most popular full-featured DB.
- Concurrency, transactions.
- Broad SQL support.
- Open source.

- Closed source.
- 🌈 "enterprise ready" 🍀
- No that doesn't actually mean anything.
- Used for historical reasons.
- Or by large orgs that need support contracts.

ON-DISK KEY-VALUE STORES

- A hash map that flushes to disk.
- Extremely simple data model.
- Uses a log-structured file for storage.
- Mostly used for data-persistence-as-a-service by more complex APIs.
- Examples: Bitcask, Apache Zookeeper

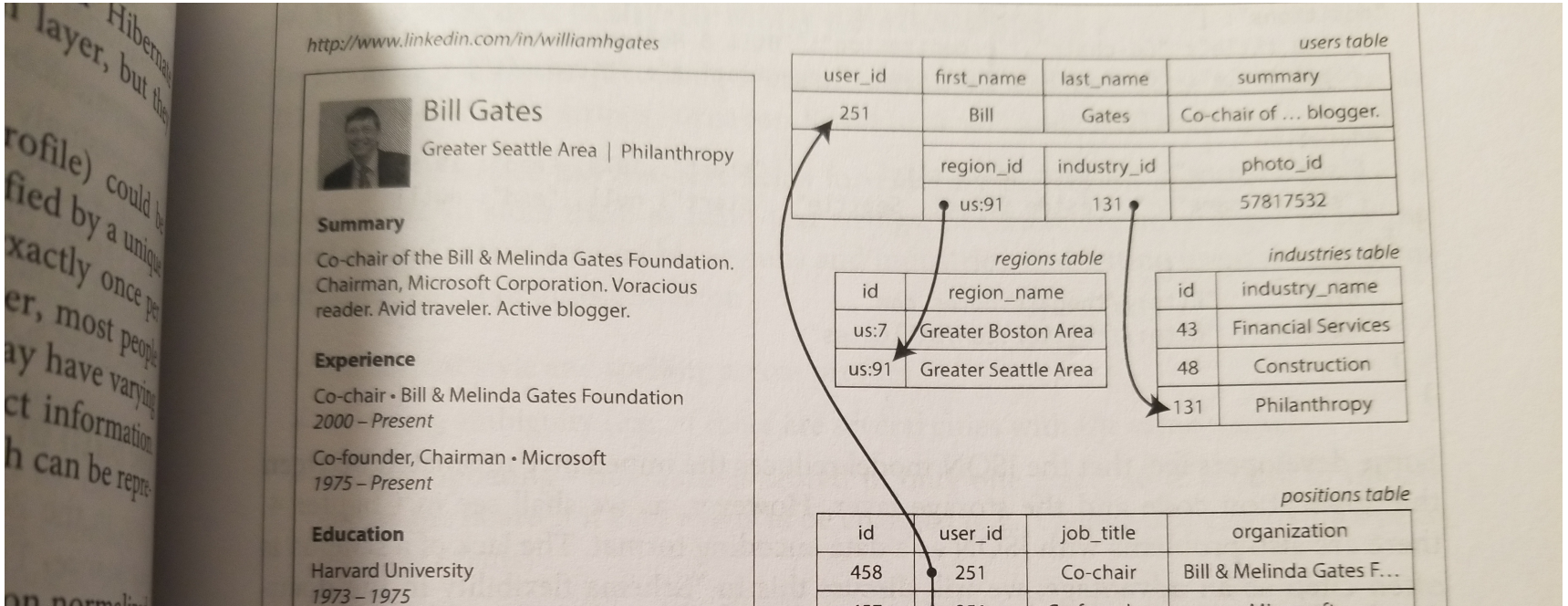
IN-MEMORY KEY-VALUE STORES

- A hash map (mostly string-to-string) that is kept in memory (RAM).
- Upside: *fast as all hell*.
- Downside: *data gets wiped on process crashes*.
- Modern implementations have some persistence features but this is a huge slow-down.
- Typically used as a **caching** layer in front of a truly

DOCUMENT STORES

- Stores data in **documents** (typically arbitrary JSON) instead of tables.
- Upside: easy to use! Promotes **locality**.
- Downside: data becomes **schema-on-read**. Limited transaction support. Implementations have historically been buggy as hell.
- Avoid using at scale.

Let's talk about **locality**. Look at those joins!



http://www.linkedin.com/in/williamhgates

Bill Gates

Greater Seattle Area | Philanthropy

Summary

Co-chair of the Bill & Melinda Gates Foundation. Chairman, Microsoft Corporation. Voracious reader. Avid traveler. Active blogger.

Experience

Co-chair • Bill & Melinda Gates Foundation
2000 – Present

Co-founder, Chairman • Microsoft
1975 – Present

Education

Harvard University
1973 – 1975

users table

user_id	first_name	last_name	summary
251	Bill	Gates	Co-chair of … blogger.

region_id	industry_id	photo_id
us:91	131	57817532

regions table

id	region_name
us:7	Greater Boston Area
us:91	Greater Seattle Area

industries table

id	industry_name
43	Financial Services
48	Construction
131	Philanthropy

positions table

id	user_id	job_title	organization
458	251	Co-chair	Bill & Melinda Gates F…

Using the document model instead, **we can store and retrieve the profile as a single object**:

```
{
    "user_id": 251,
    "first_name": "Bill",
    "last_name": "Gates",
    "summary": "Co-chair of...",
    "region_id": "us:91",
    "industry_id": 131,
    "photo_url": "/p/7/000/253/05b/308ddd6e.jpg",
    "positions": [
        {"job_title": "Co-chair", "org": "..."},
        {"job_title": "Co-founder", "org": "Microsoft"}
    ],
    "education": [
        {"school_name": "Harvard", "start": 1973, "end": 1975}
        {"school_name": "Lakeside", "start": null, "end": null
```

Consuming this in the front-end is trivially easy:

```
class LinkedInProfilePage extends React.Component {
    render() => <Profile {...mongo.getDocumentById(251)} />

}
```

This is why document stores have a reputation for being easy to use.

Let's talk about **schema-on-read**.

You insert:

```
{
    "name": "Aleksey Bilogur",

    "description": "Generally awesome."
}
```

And then:

```
{
    "name": "Joe Biden",
    "description": "He's the president!"
}
```

You write some code that queries it:

```
for person in people:
    print(person['name'])
```

Later, someone else inserts:

```
{
    "first": "Sonic",
    "last": "Hedgehog",


    "description": "Gotta go fast."
}
```

This would fail in a relational DB, but it's OK in a document store, because we don't track the document schema.

There's a problem though. Now your preexisting code will fail!

```
for person in people:
    print(person['name'])
# raises b/c the new record doesn't have a 'name' field
```

You'll have to duck test:

```
for person in people:
    if 'name' in person:
        print(person['name'])
    else:
        print(person['first'] + ' ' + person['last'])
```

Schema-on-read moves enforcing constraints from your *database* to your *application*.

In large codebases with complex documents this quickly becomes a nightmare.

"Implementations have been buggy as hell" requires a lot more exposition than I have space for here.

Suffice to say, **document stores have a reputation for silently losing documents when used at scale**.

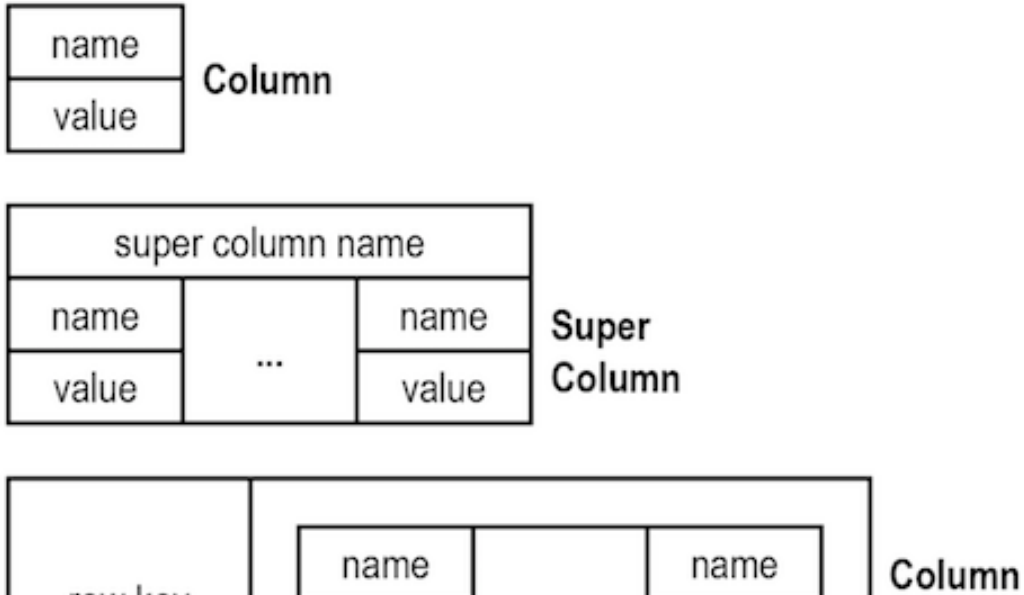
It turns out they are really hard to implement well!

For more details read Jepsen's MongoDB Review.

TLDR: do not ever use a document database as your primary database of record.

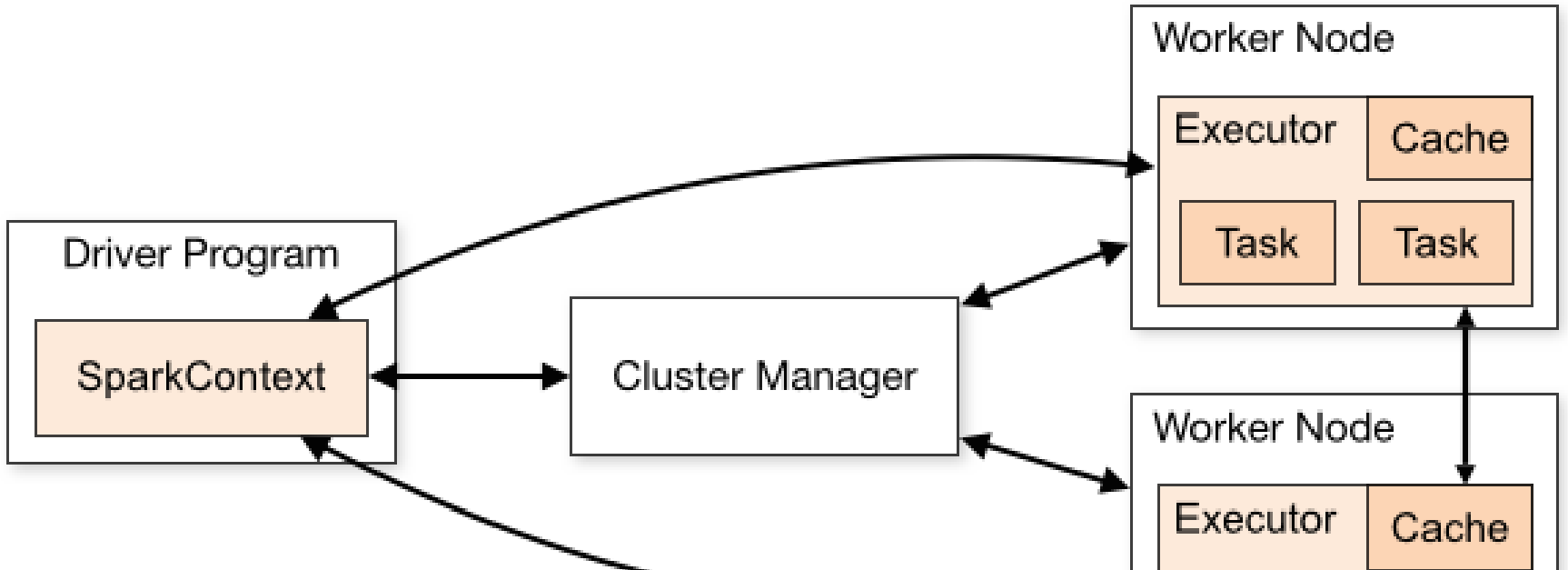
WIDE-COLUMN STORES

- Two-dimensional key-value store.
- Halfway between a relational database and a document store.
- More flexible than the relational model, more scalable than the document store model.
- Examples: Apache Cassandra, Google Bigtable.


name
value
Column
super column name
name
value
...
name
value
Super
Column
name
name
Column

COLUMN-ORIENTED STORE

- Column-oriented instead of row-oriented.
- Optimized for OLAP workloads.
- Typically use **offline storage** and **cluster compute**.
- Examples: Google BigQuery, Apache Spark (Parquet), Snowflake.


Worker Node
Executor
Cache
Task
Task
Driver Program
SparkContext
Cluster Manager
Worker Node
Executor
Cache

Some example Spark queries.

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.

Column values:

product_sk:	69	69	69	69	74	31	31	31	31	29	30	30	31	31	31	68	69	69

Bitmap for each possible value:

product_sk = 29:	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0
product_sk = 30:	0	0	0	0	0	0	0	0	0	0	1	1	0	0	0	0	0	0
product_sk = 31:	0	0	0	0	0	1	1	1	1	0	0	0	1	1	1	0	0	0
product_sk = 68:	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0
product_sk = 69:	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	1	1
product_sk = 74:	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0

Run-length encoding:

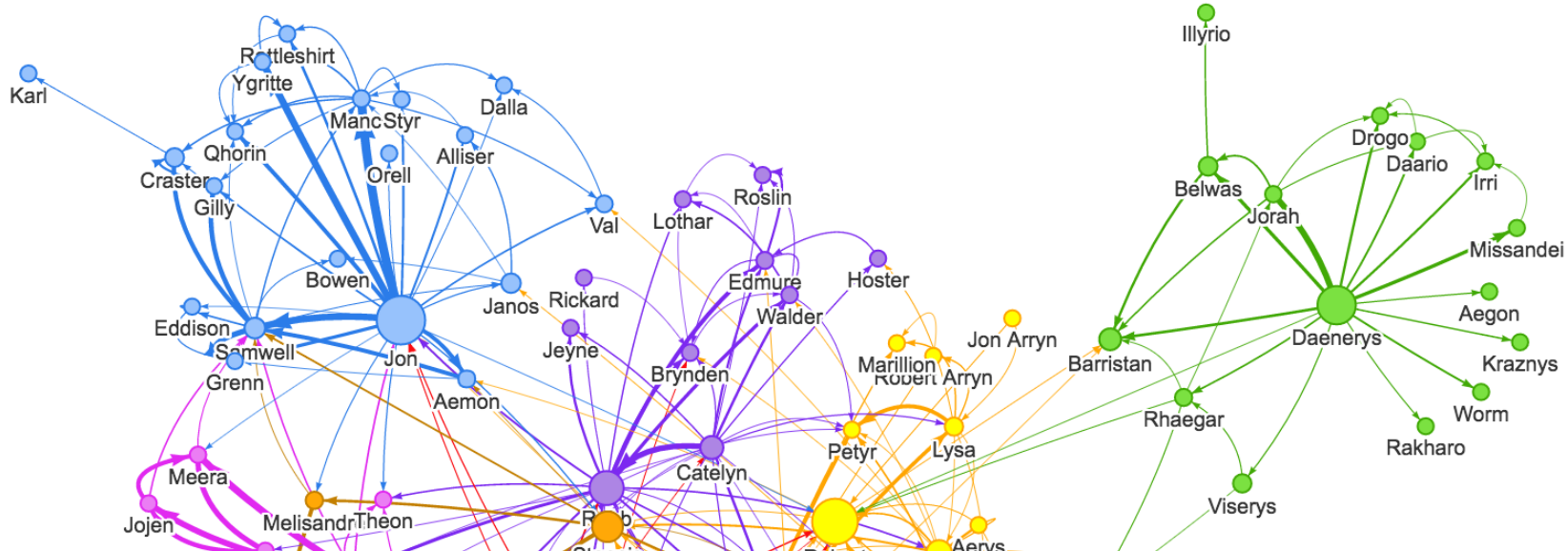
product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk = 74:	4, 1	(4 zeros, 1 one, rest zeros)

Figure 3-11. Compressed, bitmap-indexed storage of a single column.

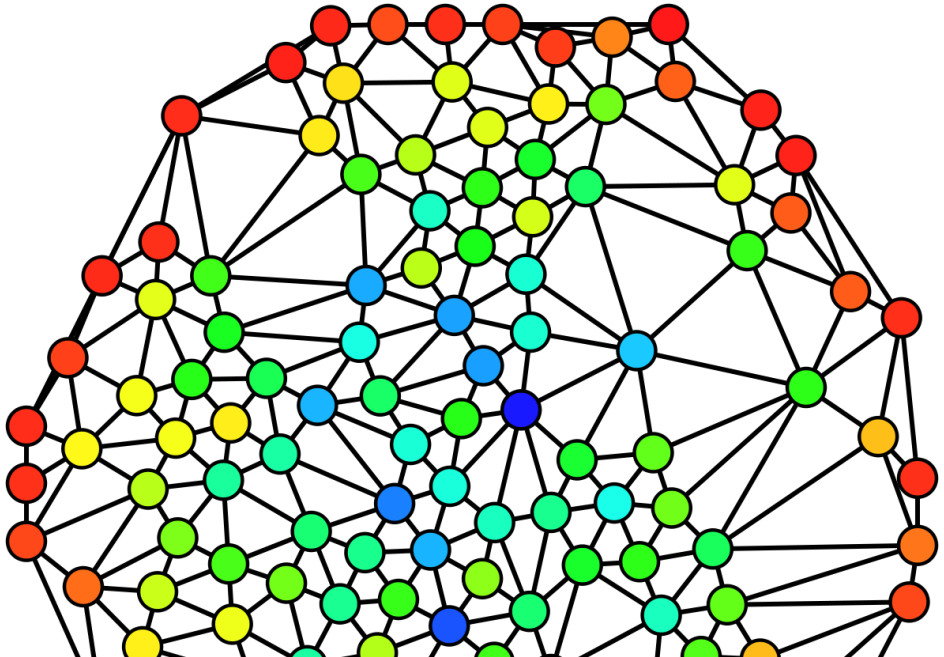
Often, the number of distinct values in a column is small compared

GRAPH DATABASES

- Optimized for graph queries (think social networks).
- Don't use SQL for queries, use SQARQL or one of its alternatives instead.
- Examples: JanusGraph, Neo4j.


Illyrio
Drogo
Daario
Irri
Belwas
Jorah
Missandei
Aegon
Daenerys
Kraznys
Worm
Rakharo
Rhaegar
Barristan
Viserys
Karl
Rattleshirt
Ygritte
Manc Styr
Dalla
Alliser
Qhorin
Orell
Craster
Gilly
Val
Lothar
Roslin
Bowen
Janos
Rickard
Edmure
Hoster
Walder
Eddison
Samwell
Jon
Jeyne
Brynden
Jon Arryn
Marillion
Robert Arryn
Grenn
Aemon
Petyr
Lysa
Meera
Catelyn
Jojen
Melisandr
Theon
Aerys

Betweeness centrality, for example.



Example SPARQL query for:

"What are all the country capitals in Africa?"

```
PREFIX ex: <http://example.com/exampleOntology#>
SELECT ?capital
       ?country
WHERE
  {
    ?x  ex:cityname        ?capital   ;
        ex:isCapitalOf     ?y         .
    ?y  ex:countryname     ?country   ;
        ex:isInContinent   ex:Africa  .
  }
```

"OTHER"

- This is my cop-out category, heh.
- This is not an exhaustive list of data models, it's just all of the ones that AFAIK are popular *right now*.
- However there are data models for niche use cases, e.g. Apache Druid or TrailDB.
- There are non-DBs with DB features, e.g. Kafka.

relational

column-oriented stores

in-memory kv stores

actual usage

document stores

wide-column stores

graph