CM3-Computer Science

What is data engineering

Data engineering is the development, implementation, and maintenance of systems and processes that take in raw data and produce high-quality, consistent information that supports downstream use cases, such as analysis and machine learning. Data engineering is the intersection of security, data management, DataOps, data architecture, orchestration, and software engineering. A data engineer manages the data engineering lifecycle, beginning with getting data from source systems and ending with serving data for use cases, such as analysis or machine learning.

-Fundamentals of Data Engineering

Our Sessions

Lecture	Big Idea
1. What is the essence of a tree?	Approach a problem at the right level of abstraction
2. Functions can travel first class too.	Idempotent functions and Functional Programming
3. Do not change things to make them faster.	Unchangedness and Immutability of data
4. Your state of mind is not my state of mind.	Modeling the state of objects with Object-Oriented Programming
5. If it looks and quacks like a duck, it is a duck.	Polymorphism and well defined interfaces.
6. If they cant eat bread, let them eat trees.	How the choice of the right data structure leads to performant algorithms.
7. Any problem can be solved by adding one extra layer of indirection	Separate the specification of an algorithm from its implementation
8. You may not use the bathroom unless I give you the key!	Concurrency and Parallelism in programs
9. Floating down a lazy river.	Lazy Evaluation and Streams
10. Deforming Mona Lisa with a new tongue.	Languages for stratified design
11. It’s turtles all the way down!	Languages and the machines which run them
12. Feed me Seymour, and I’ll remember you!	GPUs enable the learning instead of the writing of programs
13. Hey Markov! Whats the next word?	Large Language Models generate the next token


1. Whst is the essance of a tree?
3. the right lev of tsesm
2. Aporgechee things to
7 rrat lse or dase? . .
5 Opcideeren ing ttablam to
5. raiing ttsesprasitlol dostcation
8 Idempront fuctcries
8 Utchamange is fo hat
7. perrenidide prifacre.
8. Your robs chamg sres
7 ne nil-odfinfrd alliaiscions.
9 Mowl the bobtis an of icking
1. m abking my ecia prerearations.
1. Oolting the calsh lo foschiing
7 be ceiing sasnegle eres duk.
1 vick a porfiast eade.
2. Appombent fucnns
5. Urchaggdantt trake is nott
5 make ct dinrellulaticle. .
10. Hmy problemn can solved
6 anckking leria lirfractes. .
3. Dlempotrent futrees
4 Uneaianet fncions
8 Polanynlom
5 nif is me tate l of ducks
8. Plyocinste a a duck
1 Umphonem and docks
1 and ir a dees
9 Undaichoct is 'ana olind
7 fakrats orfijten duck.
10 Parcflamgle or bucs.
7 lfs the alse, opueles.
0 1028-5220? 8

What is a relational database?

- a relation (table) is a collection of tuples. Each tuple is called a *row*
- a database is a collection of tables related to each other through common data values.
- Everything in a column is values of one attribute
- A cell is expected to be atomic, no lists, dictionaries, etc
- Tables are related to each other if they have columns called keys which represent the same values
- SQL a declarative model: a query optimizer decides how to execute the query (if a field range covers 80% of values, should we use the index or the table?). Also parallelizable

How would you model data?

- The needs of OLTP databases are very different from those of OLAP databases
- OLTP databases usually need CRUD operations: CReate, Update, Delete
- OLTP tables (and incoming OLAP schemas) have a star like structure. *Fact tables* with pointers, or **keys** to *dimension tables*.
- Normalization: *The attributes of a table should be dependent on the primary key, on the whole key and nothing but the key*.

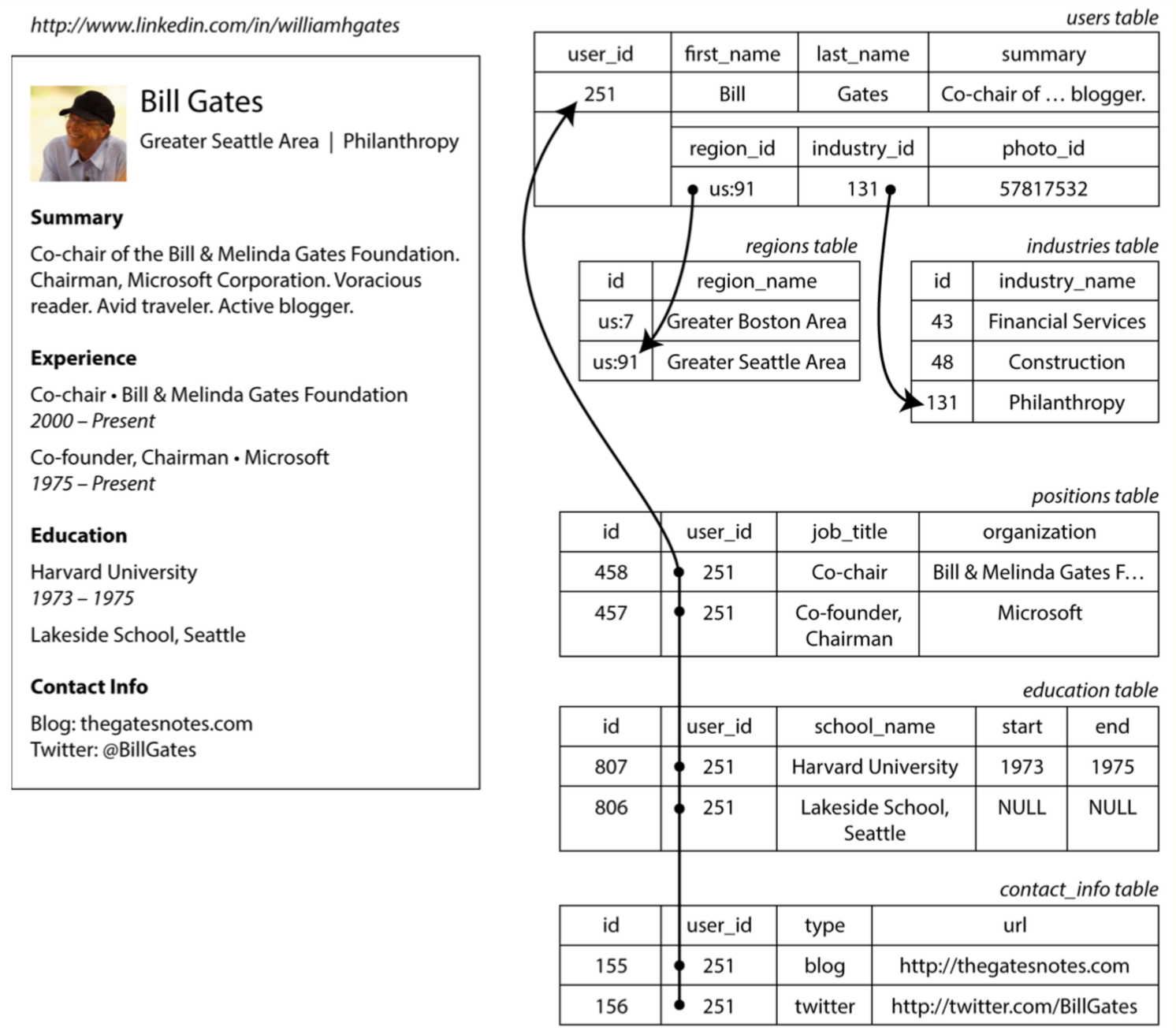


Table: contributors

New Record | Delete Record

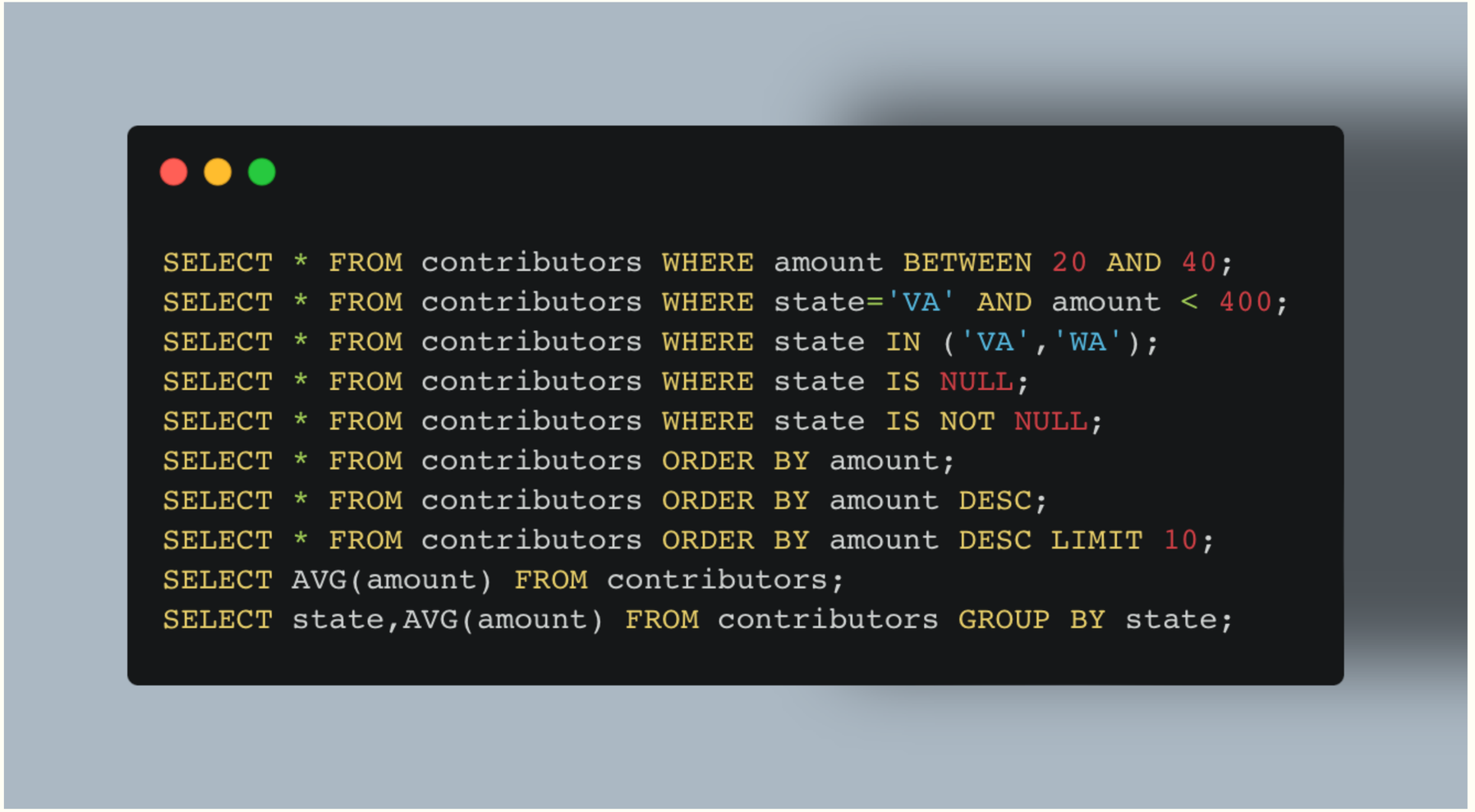
	id	last_name	first_name	middle_name	street_1	street_2	city	state	zip	amount	date	candidate_id
	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter	Filter
1	1	Agee	Steven	NULL	549 Laurel ...	NULL	Floyd	VA	24091	500	2007-06-30	16
2	5	Akin	Charles	NULL	10187 Suga...	NULL	Bentonville	AR	72712	100	2007-06-16	16
3	6	Akin	Mike	NULL	181 Baywo...	NULL	Monticello	AR	71655	1500	2007-05-18	16
4	7	Akin	Rebecca	NULL	181 Baywo...	NULL	Monticello	AR	71655	500	2007-05-18	16
5	8	Aldridge	Brittni	NULL	808 Capitol...	NULL	Washington	DC	20024	250	2007-06-06	16
6	9	Allen	John D.	NULL	1052 Cann...	NULL						
7	10	Allen	John D.	NULL	1052 Cann...	NULL						
8	11	Allison	John W.	NULL	P.O. Box 10...	NULL						
9	12	Allison	Rebecca	NULL	3206 Sum...	NULL						

	id	first_name	last_name	middle_name	party
	Filter	Filter	Filter	Filter	Filter
1	16	Mike	Huckabee		R
2	20	Barack	Obama		D
3	22	Rudolph	Giuliani		R
4	24	Mike	Gravel		D
5	26	John	Edwards		D
6	29	Bill	Richardson		D
7	30	Duncan	Hunter		R
8	31	Dennis	Kucinich		D
9	32	Ron	Paul		R

Create Tables

```sql
DROP TABLE IF EXISTS "candidates";
DROP TABLE IF EXISTS "contributors";
CREATE TABLE "candidates" (
    "id" INTEGER PRIMARY KEY  NOT NULL ,
    "first_name" VARCHAR,
    "last_name" VARCHAR,
    "middle_name" VARCHAR,
    "party" VARCHAR NOT NULL
);
CREATE TABLE "contributors" (
    "id" INTEGER PRIMARY KEY  AUTOINCREMENT  NOT NULL,
    "last_name" VARCHAR,
    "first_name" VARCHAR,
    "middle_name" VARCHAR,
    "street_1" VARCHAR,
    "street_2" VARCHAR,
    "city" VARCHAR,
    "state" VARCHAR,
    "zip" VARCHAR, -- Notice that we are converting the zip from integer to string
    "amount" INTEGER,
    "date" DATETIME,
    "candidate_id" INTEGER NOT NULL,
    FOREIGN KEY(candidate_id) REFERENCES candidates(id)
);
```

selects in tables

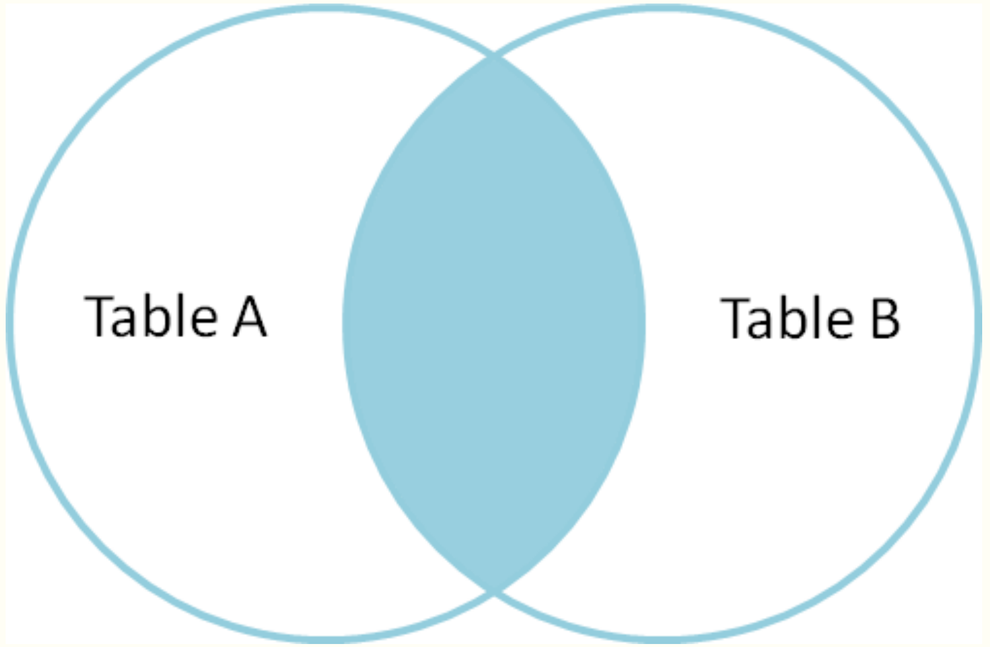


```sql
SELECT * FROM contributors WHERE amount BETWEEN 20 AND 40;
SELECT * FROM contributors WHERE state='VA' AND amount < 400;
SELECT * FROM contributors WHERE state IN ('VA','WA');
SELECT * FROM contributors WHERE state IS NULL;
SELECT * FROM contributors WHERE state IS NOT NULL;
SELECT * FROM contributors ORDER BY amount;
SELECT * FROM contributors ORDER BY amount DESC;
SELECT * FROM contributors ORDER BY amount DESC LIMIT 10;
SELECT AVG(amount) FROM contributors;
SELECT state,AVG(amount) FROM contributors GROUP BY state;
```

inserts and alters

```
INSERT INTO candidates (id, first_name, last_name, middle_name, party) VALUES (?,?,?,?,?);
ALTER TABLE contributors ADD COLUMN name;
ALTER TABLE contributors DROP COLUMN name;
DELETE FROM contributors WHERE last_name="Ahrens";
drop table if exists mailing_list;

create table mailing_list (
        email           varchar(100) not null primary key,
        name            varchar(100)
);

drop table if exists phone_numbers;
create table phone_numbers (
        email           varchar(100) not null references mailing_list(email),
        number_type     varchar(15) check (number_type in ('work','home','cell','beeper')),
        phone_number    varchar(20) not null
);
insert into phone_numbers values ('ogrady@fastbuck.com','work','(800) 555-1212');
insert into phone_numbers values ('ogrady@fastbuck.com','home','(617) 495-6000');
insert into phone_numbers values ('philg@mit.edu','work','(617) 253-8574');
insert into phone_numbers values ('ogrady@fastbuck.com','beeper','(617) 222-3456');
```

Inner Joins



left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

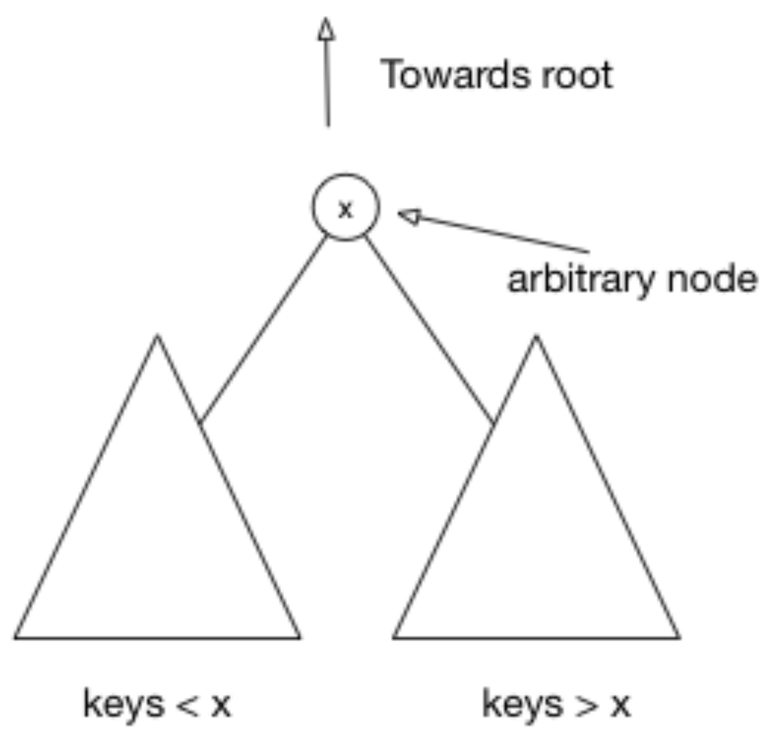
	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

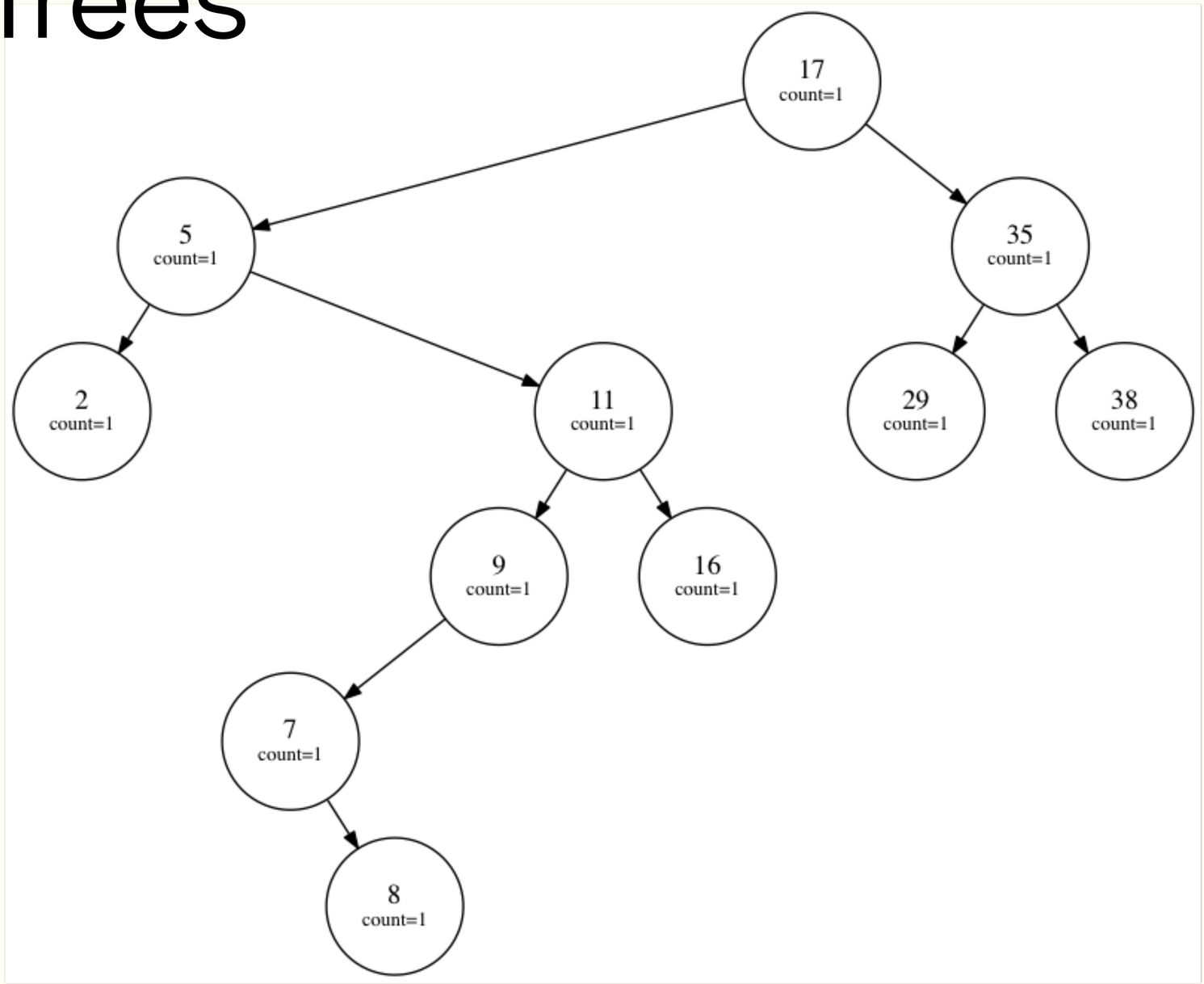
Storage Components of a RDBMS

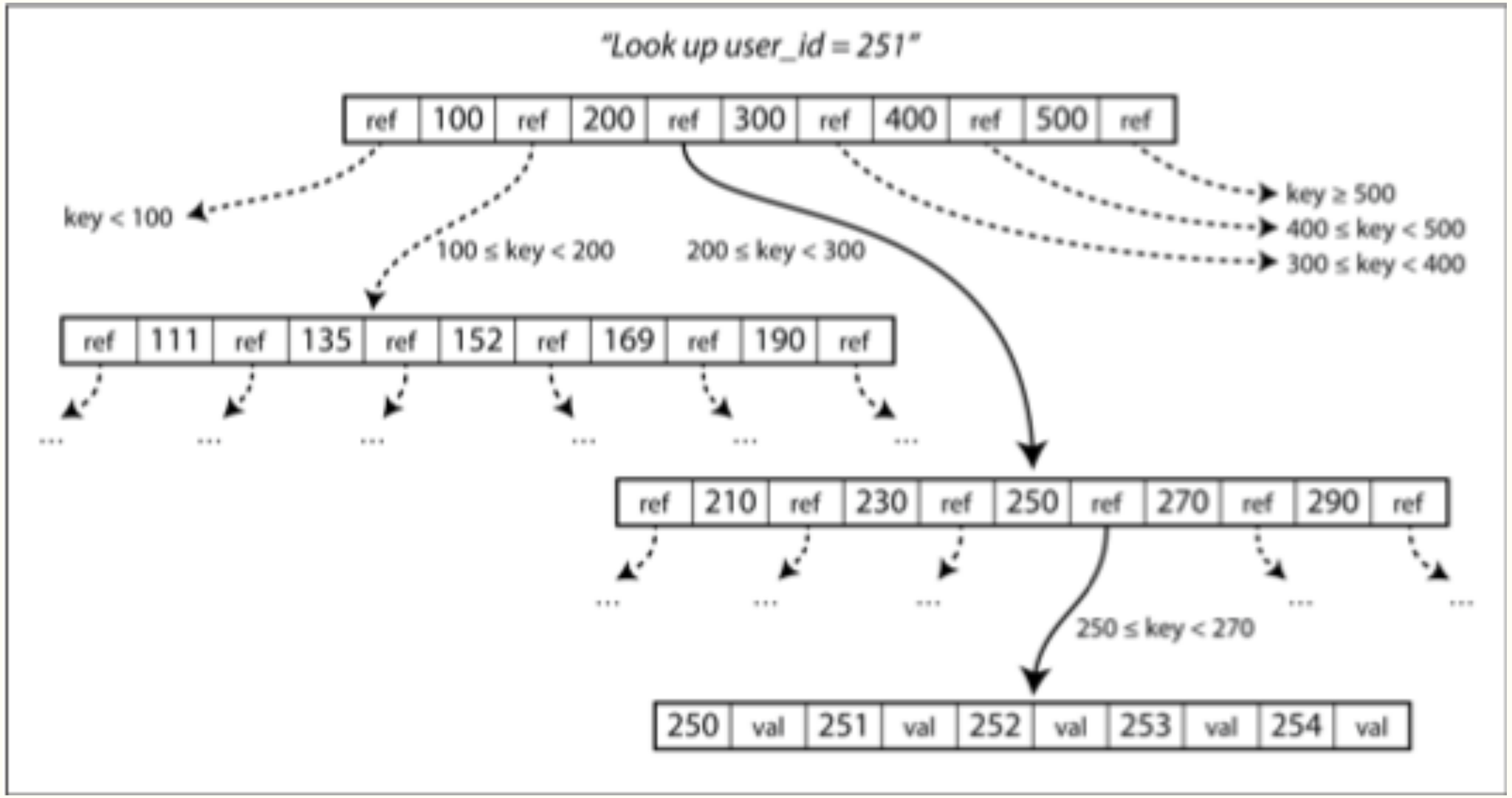
- the heap file: this is where the rows or columns are stored
- regular relational databases use row oriented heap files
- an index file(s): this is where the index for a particular attribute is stored
- sometimes you have a clustered index (all data stored in index) or covering index (some data is stored in index)
- the WAL or write-ahead log: this is used to handle transactions

Binary Search Trees

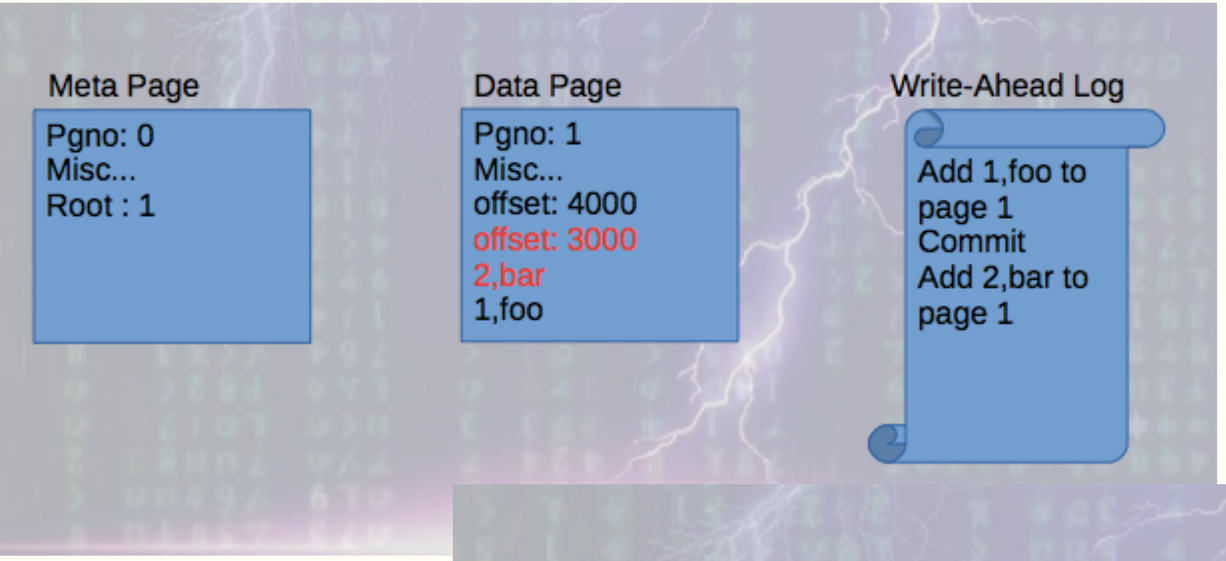
- These have the property that for any value x, numbers less go to the left and numbers right go to the right
- Consider a list: [17,5,35,2,11,29,38,9,16,7,8]
- Then the binary tree you get is:

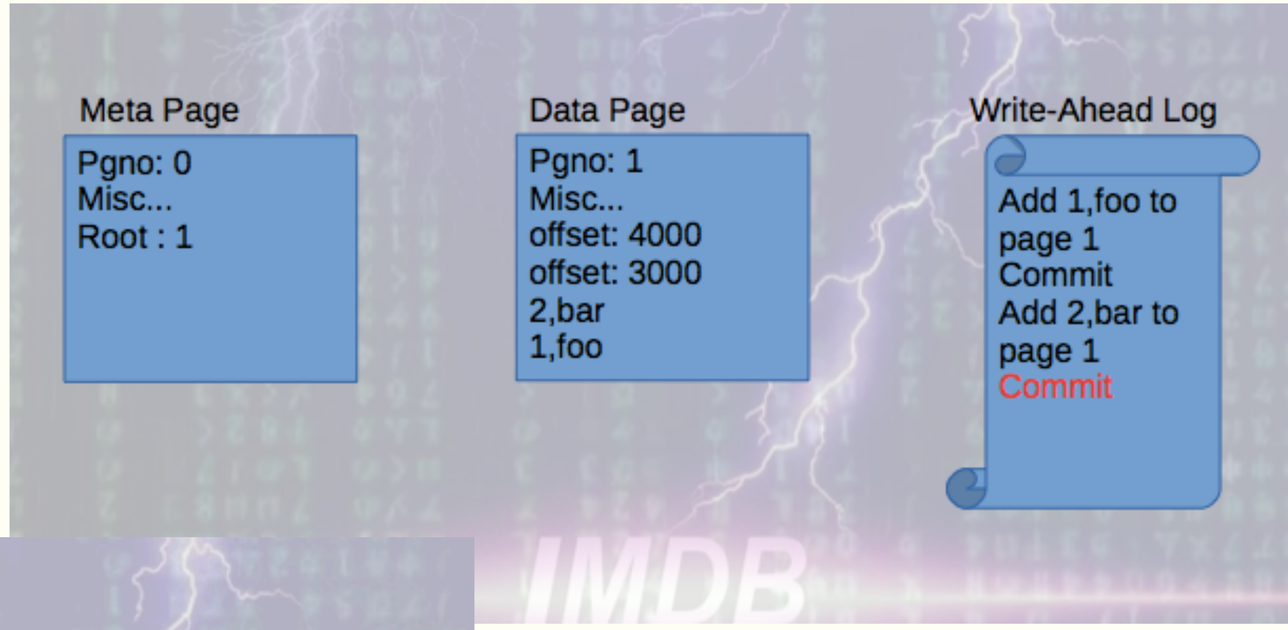


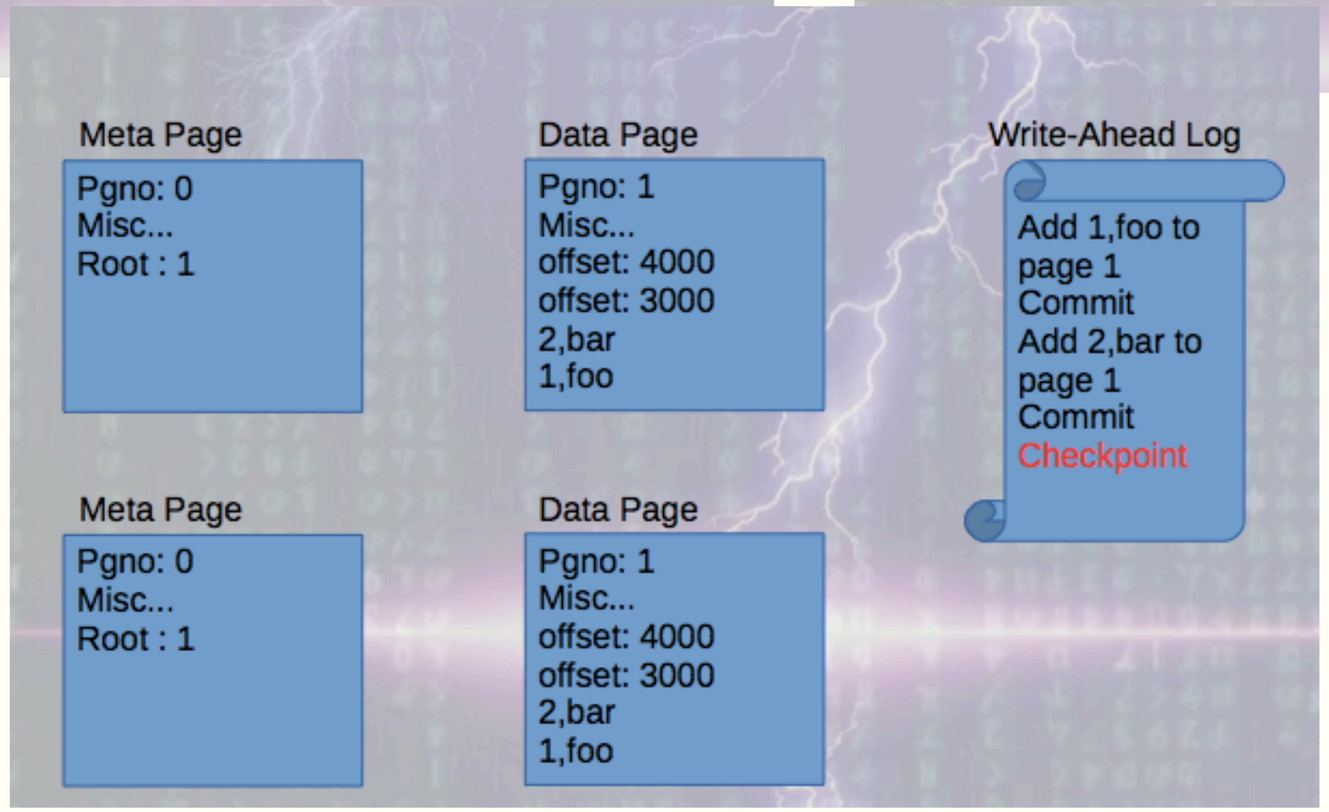



"Look up user_id = 251"
ref 100 ref 200 ref 300 ref 400 ref 500 ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
300 ≤ key < 400
400 ≤ key < 500
key ≥ 500
ref 111 ref 135 ref 152 ref 169 ref 190 ref
ref 210 ref 230 ref 250 ref 270 ref 290 ref
250 ≤ key < 270
250 val 251 val 252 val 253 val 254 val

Btrees


Meta Page
Pgno: 0
Misc...
Root : 1
Data Page
Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo
Write-Ahead Log
Add 1,foo to page 1
Commit
Add 2,bar to page 1


Meta Page
Pgno: 0
Misc...
Root : 1
Data Page
Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo
Write-Ahead Log
Add 1,foo to page 1
Commit
Add 2,bar to page 1
Commit
IMDB


Meta Page
Pgno: 0
Misc...
Root : 1
Data Page
Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo
Write-Ahead Log
Add 1,foo to page 1
Commit
Add 2,bar to page 1
Commit
Checkpoint
Meta Page
Pgno: 0
Misc...
Root : 1
Data Page
Pgno: 1
Misc...
offset: 4000
offset: 3000
2,bar
1,foo

WAL

General requirements for e-commerce

- The batch of operations is viewed as a single atomic operation, so all of the operations either succeed together or fail together.
- The database is in a valid state before and after the transaction.
- The batch update appears to be isolated; no other query should ever see a database state in which only some of the operations have been applied.

Transactions: ACID

- A is for **atomicity**. The batch of operations is viewed as a single atomic operation, so all of the operations either succeed together or fail together. This means that the batch of operations either all happen (**commit**) or not happen at all (**abort**, **rollback**).

- The batch update appears to be **isolated**; other queries should never see a database state in which only some of the operations have been applied. I is for Isolation. This *is the most interesting of the lot*, and critical to the sensible running of a database. The idea is that transactions should not step on each other. Each transaction should pretends that its the only one running on the database: in other words, as if the transactions were completely serialized.

- In practice this would make things very slow, so we try different transactional guarantees that fall short of explicit serialization except in the situations that really need serialization.

- The database is in a valid state before and after the transaction. D is for **Durability**: once a transaction has comitted successfully, data comitted wont be forgotten. This requires persistent storage, or replication, or both.

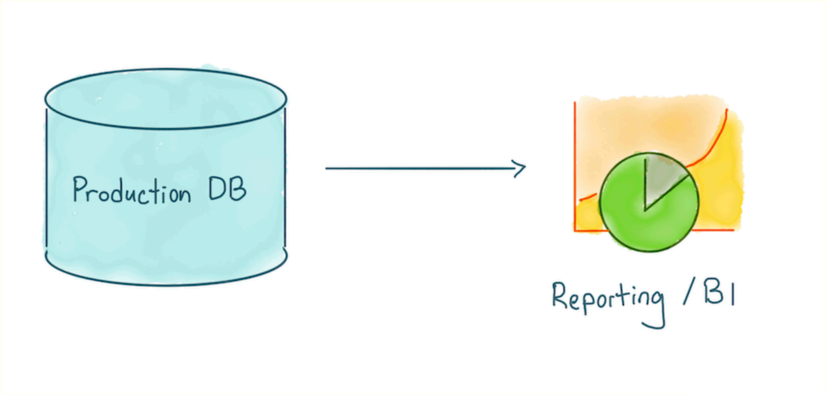
- C is for **Consistency**: data invariants must be true. This is really a property of the application: eg accounting tables must be balanced. Databases can help with foreign keys, but this is a property of the app. We wont discuss this one further.

Why is isolation important?

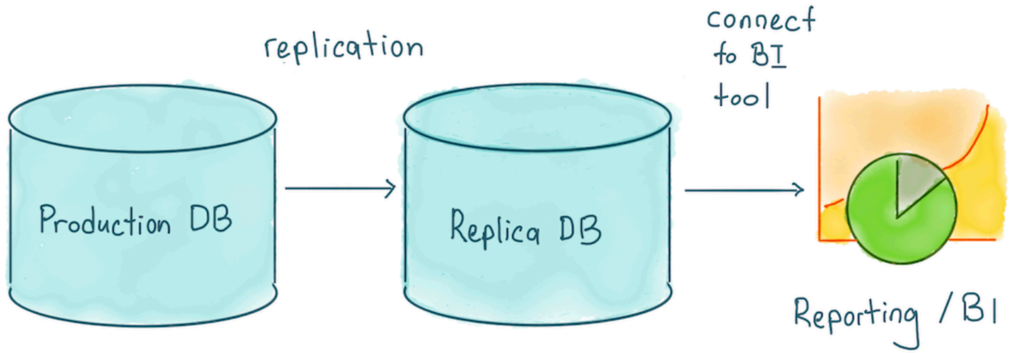
- It is hard to program without isolation. isolation is what guarantees stability. It is what makes sure that there are no dirty reads and dirty writes.
- **Dirty read**s: One client reads another client's writes before they have been committed. This could mean you see a value that would be later rolled back.
- **Dirty writes**: One client overwrites data that another client has written, but not yet committed. Bad. When we write we will use a lock to ensure no-one else can write.
- Clearly, the notions of isolation are really the notions of concurrency: these issues will also occur when 2 programs access any data, in memory or in a database. In both cases locks and other ideas must be used to make sure that there is only one mutator at a time, and that an object is not exposed in an inconsistent state.

From transactions to analysis

Organizational Evolution

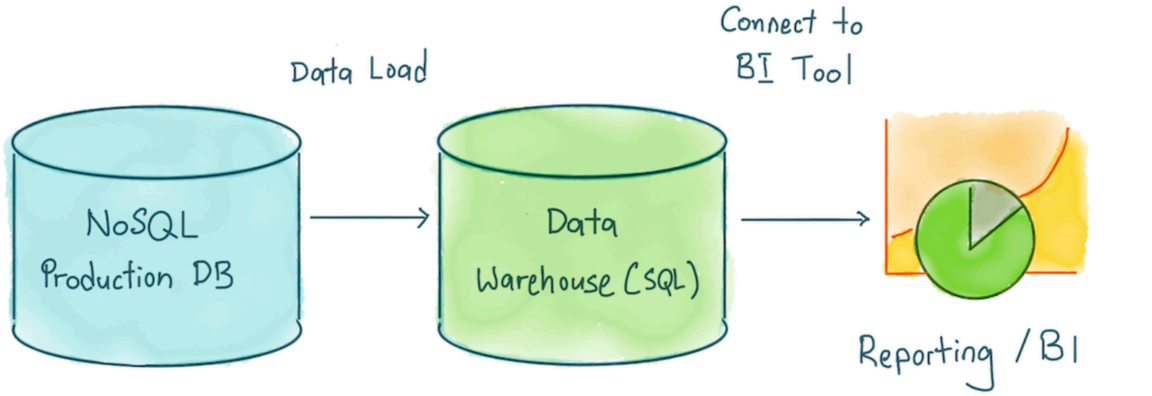


Start by using the production database as your analytics database

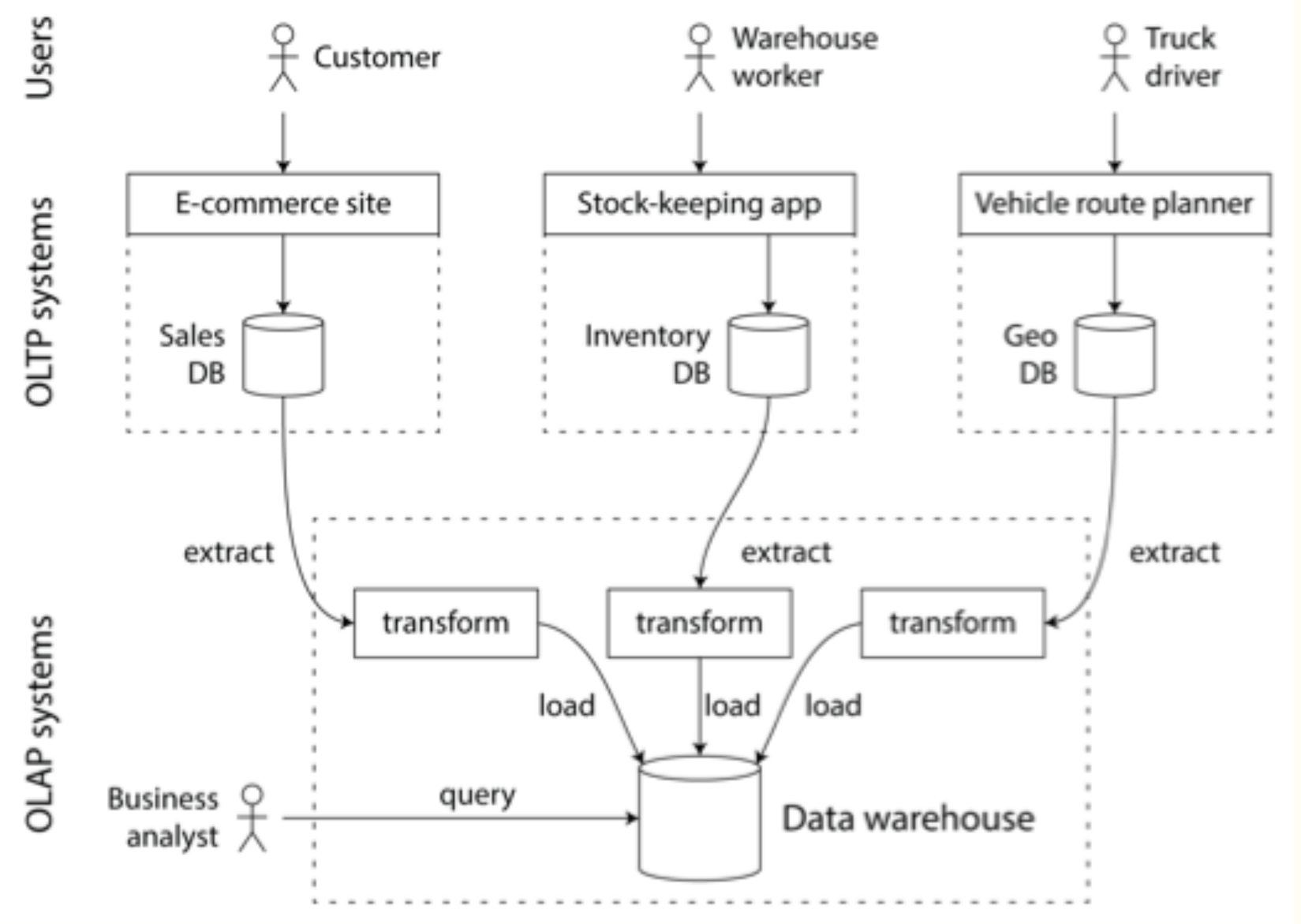


Move to using a replica of the production database as your analytics database

Multiple production databases; use a warehouse instead and not affect their operation and to enable efficient analytics



How are the databases used?



Transactional DBs vs. Analytics DBs

Data:

- Many single-row writes
- Current, single data

Queries:

- Generated by user activities; 10 to 1000 users
- < 1s response time
- Short queries

Data:

- Few large batch imports
- Years of data, many sources

Queries:

- Generated by large reports; 1 to 10 users
- Queries run for hours
- Long, complex queries

Ref: http://www.slideshare.net/PGExperts/really-big-elephants-postgresql-dw-15833438 (slide 5)

Columnar Databases

- Store each column separately
- Have a higher read efficiency as only a few columns of contiguous or run-encoded data need to be read
- compress better especially if the cardinality of the columns is not high thus allowing more data to be loaded into memory
- columnar data have higher sorting and indexing efficiency ans may even admit multiple sort orders

Row vs Column

OrderId	CustomerId	ShippingCountry	OrderTotal
1	1258	US	55.25
2	5698	AUS	125.36
3	2265	US	776.95
4	8954	CA	32.16

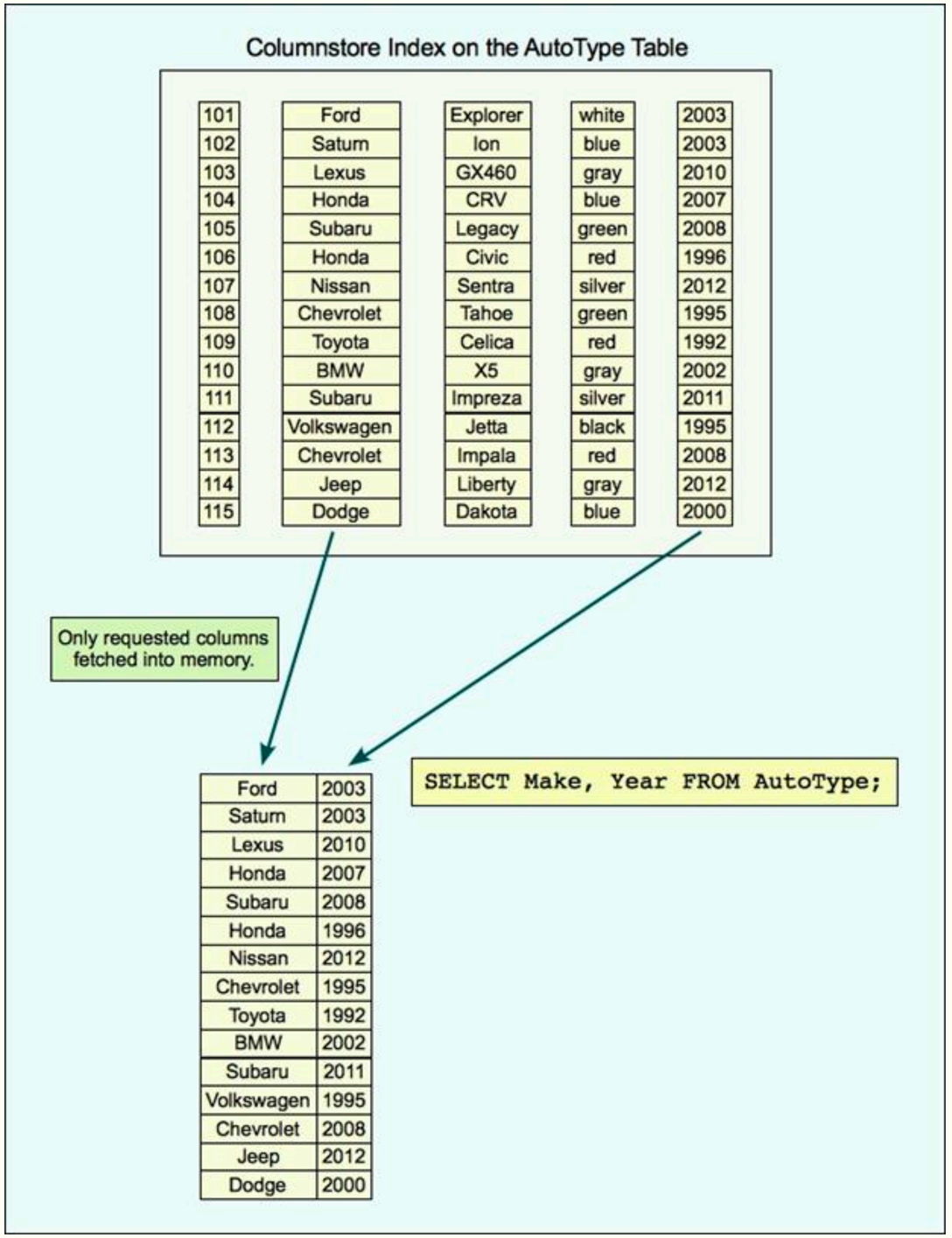
Block 1	1, 1258, US, 55.25
Block 2	2, 5698, AUS, 125.36
Block 3	3, 2265, US, 776.95
Block 4	4, 8954, CA, 32.16

OrderId	CustomerId	Shipping Country	Order Total	Customer Active
1	1258	US	55.25	TRUE
2	5698	AUS	125.36	TRUE
3	2265	US	776.95	TRUE
4	8954	CA	32.16	FALSE

Block 1	1, 2, 3, 4
Block 2	1258, 5698, 2265, 8954
Block 3	US, AUS, US, CA
Block 4	55.25, 125.36, 776.95. 32.16
Block 5	TRUE, TRUE, TRUE, FALSE

Column-oriented Storage

- store values from each column together in separate storage
- lends itself to compression with bitmap indexes
- compressed indexes can fit into cache and are usable by iterators
- several different sort orders can be redundantly stored
- writing is harder: updating a row touches many column files
- but you can write an in-memory front sorted store (row or column), and eventually merge onto the disk



Bitmap Indexes

- lends itself to compression with bitmap indexes and run-length encoding. This involves choosing an appropriate sort order. The index then can be the data (great for IN and AND queries): there is no pointers to “elsewhere"
- bitwise AND/OR can be done with vector processing

Column values:

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

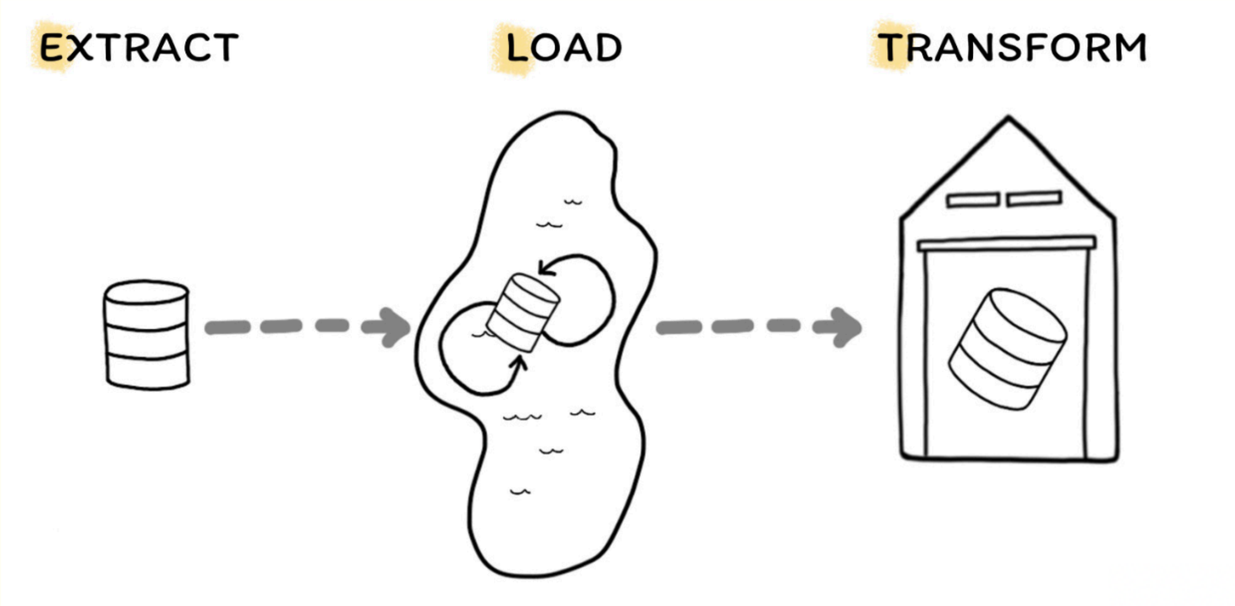
Bitmap for each possible value:

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:

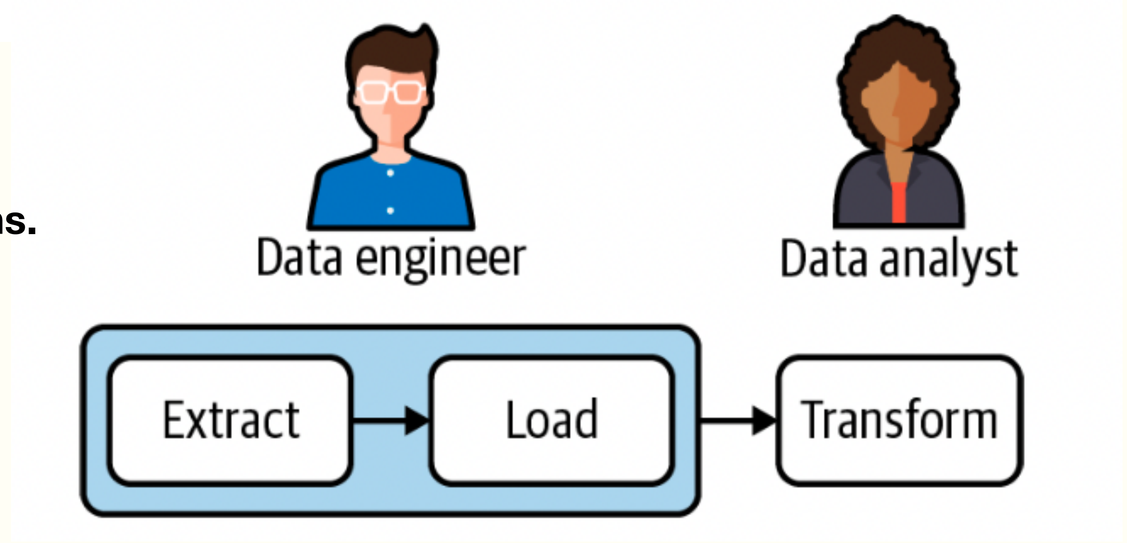
product_sk = 29:	9, 1	(9 zeros, 1 one, rest zeros)
product_sk = 30:	10, 2	(10 zeros, 2 ones, rest zeros)
product_sk = 31:	5, 4, 3, 3	(5 zeros, 4 ones, 3 zeros, 3 ones, rest zeros)
product_sk = 68:	15, 1	(15 zeros, 1 one, rest zeros)
product_sk = 69:	0, 4, 12, 2	(0 zeros, 4 ones, 12 zeros, 2 ones)
product_sk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

ETL to ELT



Move stuff into a warehouse or lake first!
This is called a source refreshed table

Now transformations are done IN the warehouse.
The data engineer can do the initial loading and transformations.
Data analysts from client groups (sales, marketing)
can do subsequent transformations



Meanwhile at Google/Yahoo/ Facebook

The data rate is too damn high….

Pure functions

- **return the same values for the same arguments**, thus being like mathematical functions,
- **no side effects**, by which some state is changed during the execution of the function, and this might not be repeatable, and
- **referential transparent**, by which one may replace the function call by the resultant return values in the code.

Gluing using pure functions: sum and product

```
def fsum(alist):
    if len(alist)==0: # empty list
        return 0
    else:
        return alist[0] + fsum(alist[1:])
```

```
fsum([])
```

```
0
```

```
fsum([1])
```

```
1
```

```
fsum([1,2,3])
```

```
6
```

The key here is to recursively call the same function again

```
def fprod(alist):
    if len(alist)==0: # empty list
        return 1
    else:
        return alist[0] * fprod(alist[1:])
```

```
fprod([]), fprod([1]), fprod([1,2]), fprod([1,2,3])
```

```
(1, 1, 2, 6)
```

Abstracting the pure function out…

```
import operator
def f_op(alist, op, startval):
    if len(alist)==0: # empty list
        return startval
    else:
        return op(alist[0], f_op(alist[1:], op, startval))
f_op([1,2,3], operator.add, 0)
```

```
6
```

```
f_op([1,2,3], operator.mul, 1)
```

```
6
```

sum and product: here f_op **reduces** to a number. Such a function is called a **foldr**

But what if we used “reduction” to construct a list?

```
def cons(a,b):
    return [a] + b
f_op([1,2,3], cons, [])
```

```
[1, 2, 3]
```

list construction

```
def double_and_cons(a, b):
    return [2*a] + b
f_op([1,2,3], double_and_cons, [])
```

```
[2, 4, 6]
```

list construction and doubling

Abstracting even further…gives us a map

```
def double_and_cons(a, b):
    return [2*a] + b
f_op([1,2,3], double_and_cons, [])
```

```
[2, 4, 6]
```

```
def f_map(f, alist):
    f = f_and_cons_composer(f)
    return f_op(alist, f, [])
```

```
f_map(double, [1,2,3])
```

```
[2, 4, 6]
```

```
double = lambda j: 2*j
def f_and_cons_composer(f):
    def f_and_cons(a, b):
        return [f(a)] + b
    return f_and_cons
double_and_cons2 = f_and_cons_composer(double)
```

```
f_op([1,2,3], double_and_cons2, [])
```

```
[2, 4, 6]
```

A function which takes a list and gives us another list of the same size is called a map.

Function Glue: Map and Foldr(reduce) plus pure functions

- Many problems can be expressed in this **map** paradigm where we map a list to another list by some pure function
- Then we **reduce** or foldr the new list to a result, again via a pure function
- We can do this over huge bits of data by splitting them up into multiple lists over multiple machines
- If a machine fails, the purity comes to our rescue. We just re-run that part of the computation with our pure function, and re-combine it in
- This **map-reduce** idea owes its existence to Gluing Functions together in the map and reduce phases.

MapReduce: Simplified Data Processing on Large Clusters

Jeffrey Dean and Sanjay Ghemawat

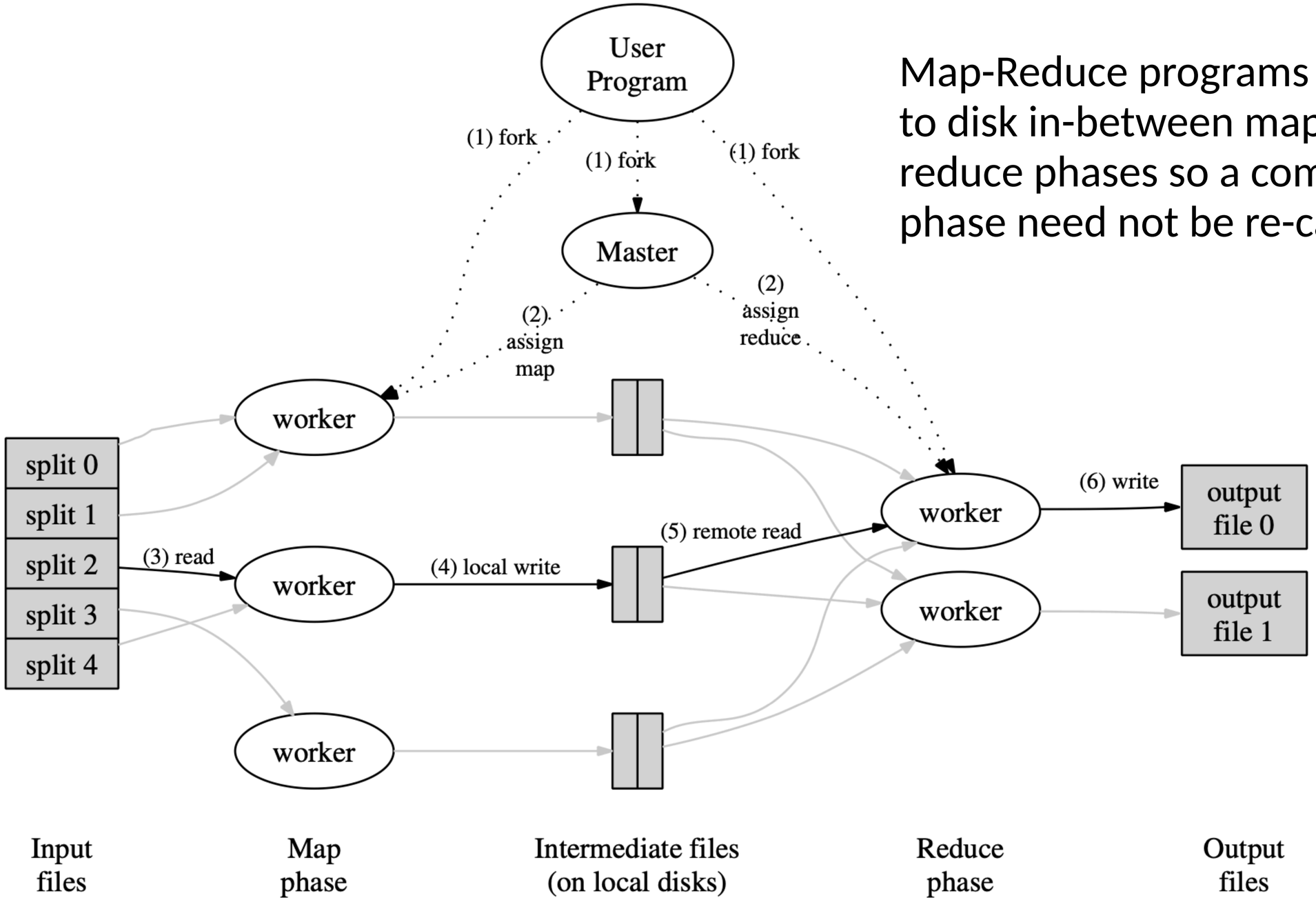
jeff@google.com, sanjay@google.com

Google, Inc.

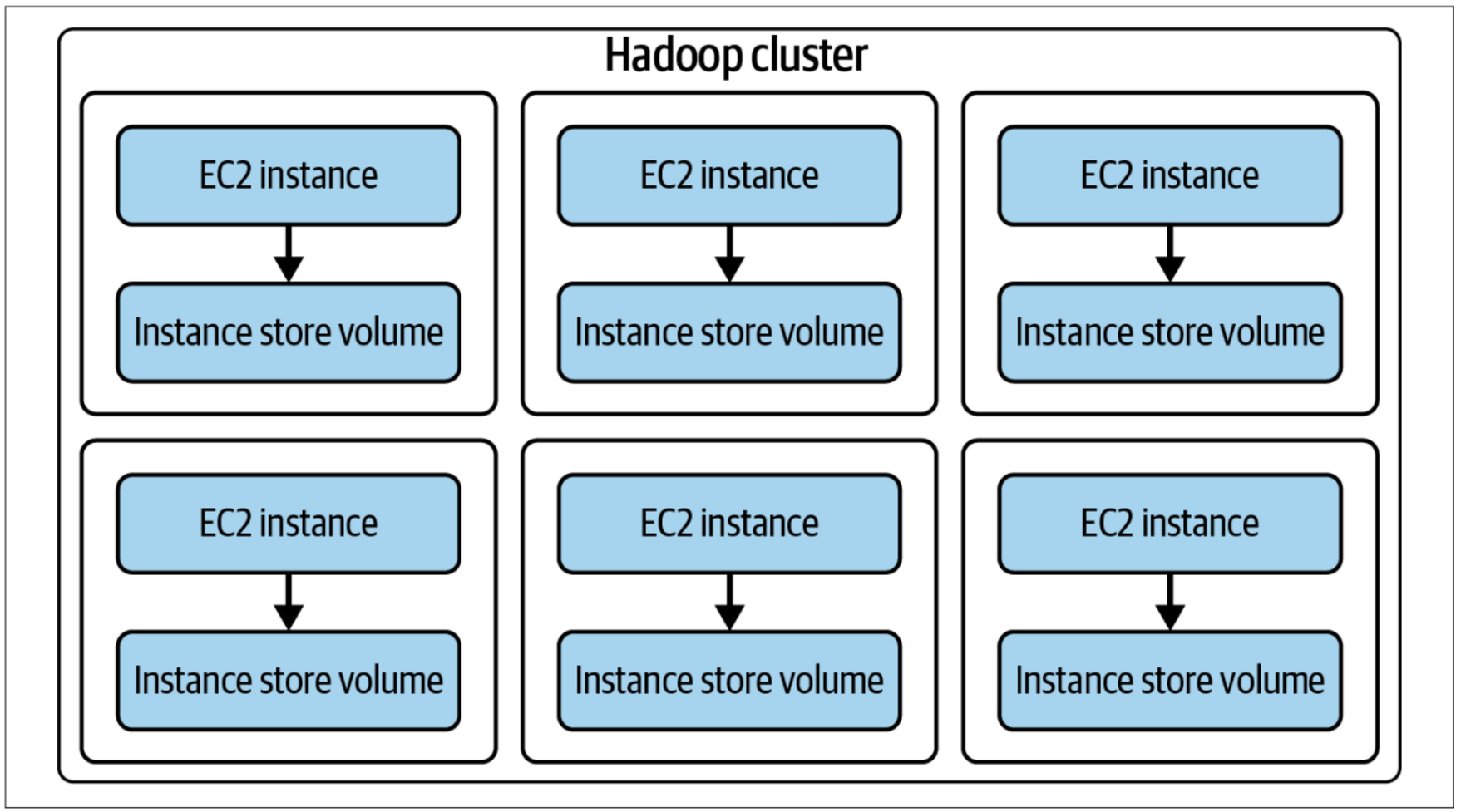
Key Idea: Pure functions glued into map and reduce functions allow us to parallelize programs and recover from cluster node failure

```
map(String key, String value):
  // key: document name
  // value: document contents
  for each word w in value:
    EmitIntermediate(w, "1");

reduce(String key, Iterator values):
  // key: a word
  // values: a list of counts
  int result = 0;
  for each v in values:
    result += ParseInt(v);
  Emit(AsString(result));
```

Map-Reduce programs write to disk in-between map and reduce phases so a complete phase need not be re-calculated.



Distributed Computing



Keys are character line offsets

The map function acts on each key, here, and extracts years and temps

```
0067011990999991950051507004...9999999N9+00001+99999999999...
0043011990999991950051512004...9999999N9+00221+99999999999...
0043011990999991950051518004...9999999N9-00111+99999999999...
0043012650999991949032412004...0500001N9+01111+99999999999...
0043012650999991949032418004...0500001N9+00781+99999999999...
```

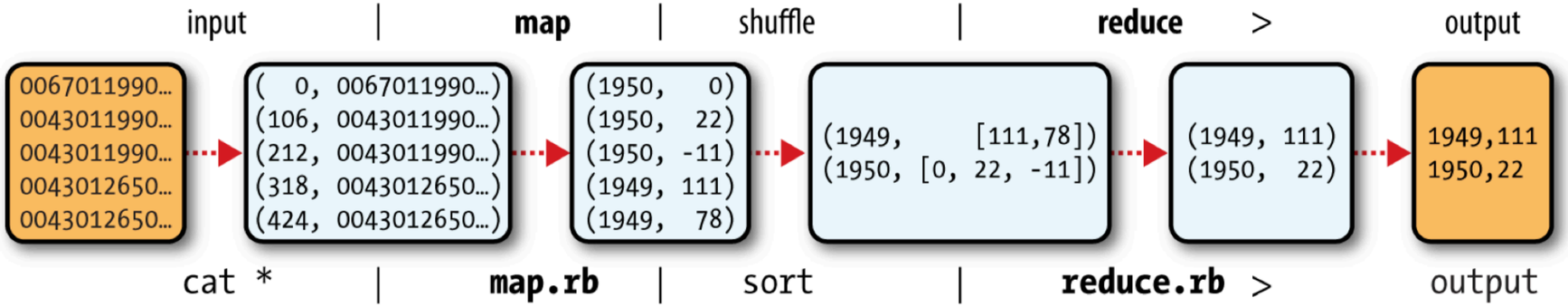
These lines are presented to the map function as the key-value pairs:

```
(0, 0067011990999991950051507004...9999999N9+00001+99999999999...)
(106, 0043011990999991950051512004...9999999N9+00221+99999999999...)
(212, 0043011990999991950051518004...9999999N9-00111+99999999999...)
(318, 0043012650999991949032412004...0500001N9+01111+99999999999...)
(424, 0043012650999991949032418004...0500001N9+00781+99999999999...)
```

```
(1950, 0)
(1950, 22)
(1950, -11)
(1949, 111)
(1949, 78)
```

The mapreduce framework collates the temps by year into a list and sorts them.

```
(1949, [111, 78])
(1950, [0, 22, -11])
```



The reduce function gets the max over the list.

```
(1949, 111)
(1950, 22)
```

Finding Square Roots by Newton's method

Newton's algorithm computes the square root of a number n by starting from an initial approximation a_0 and computing better and better ones using the rule

$$a_{i+1} = (a_i + n/a_i)/2$$

If the approximations converge to some limit a, then

$$a = (a + n/a)/2$$

so

$$\begin{aligned} 2a &= a + n/a \\ a &= n/a \\ a * a &= n \\ a &= \sqrt{n} \end{aligned}$$

```python
def square_root(n, eps, a0):
    x_prev = a0
    while True:
        assert x_prev!=0
        x_next = 0.5*(x_prev + n/x_prev)
        print(x_next)
        if abs(x_next - x_prev) < eps:
            return x_next
        else:
            x_prev = x_next
```

In fact the approximations converge rapidly to a limit. Square root programs take a tolerance (eps) and stop when two successive approximations differ by less than eps.

```python
square_root(9, 0.0001, 1)
```

```
5.0
3.4
3.023529411764706
3.00009155413138
3.000000001396984
3.000000001396984
```

Writing using pure functions

```
def good_enough_maker(eps):
    def is_it_good_enough(x_next, x_prev):
        if abs(x_next - x_prev) < eps:
            return True
        return False
    return is_it_good_enough
```

```
good_enough = good_enough_maker(0.0001)
```

```
def improve(n, x_prev):
    return 0.5*(x_prev + n/x_prev)
```

And compose a recursive routine which we can spread on a cluster ala hadoop, since these are all pure functions.

```
def sqrt_iter(guess, n):
    x_next = improve(n, guess)
    if good_enough(x_next, x_prev):
        return x_next
    else:
        return sqrt_iter(x_next, n)
```

Gluing Programs

```
def improve(n, x_prev):
    return 0.5*(x_prev + n/x_prev)
```

```
def within_maker(eps, n):
    def is_it_within(x_next, x_prev):
        if abs(x_next - x_prev) < eps:
            return x_next
        return is_it_within(improve(n, x_next), x_next)
    return is_it_within
```

```
within = within_maker(0.0001, 9)
```

```
within(improve(9, 1),1)
```

```
3.000000001396984
```

- The test for convergence now becomes the outer part of the program
- “improve” is run within it
- we still want to divorce “improve” from “within”
- this requires pausing the “improve” function when the conditions of the “within” program are met
- this turns “improve” into a pausable, or generator function. This is advanced python, but the idea is a very general one.

SPARK

We keep the pure functions and the map and reduce (**Gluing Functions**) ideas from map-reduce/hadoop. The lack of side effects and state-dependence allow us to recover from node failures.

There are key new ideas:

Immutable Data structures: **RDD**s in memory!

Keeping track of parent stage in execution, or more precisely, parent immutable data structure. The data structure for this is called a **DAG**, or Directed Acyclic Graph.

We separate functions into transformations and actions. Nothing is run on transformations, and runs only happen on actions. This is a key new functional idea called **Gluing Programs.**

- Created by Matei Zaharia et. al. at AMPLab in Berkeley.
- Open Source
- Successor to Hadoop
- Commercialized as Databricks, Inc
- Adds some key ideas to the functional programming and pure functions from Hadoop.
- Can skip disk writing
- Can skip some computations if not needed by final output.

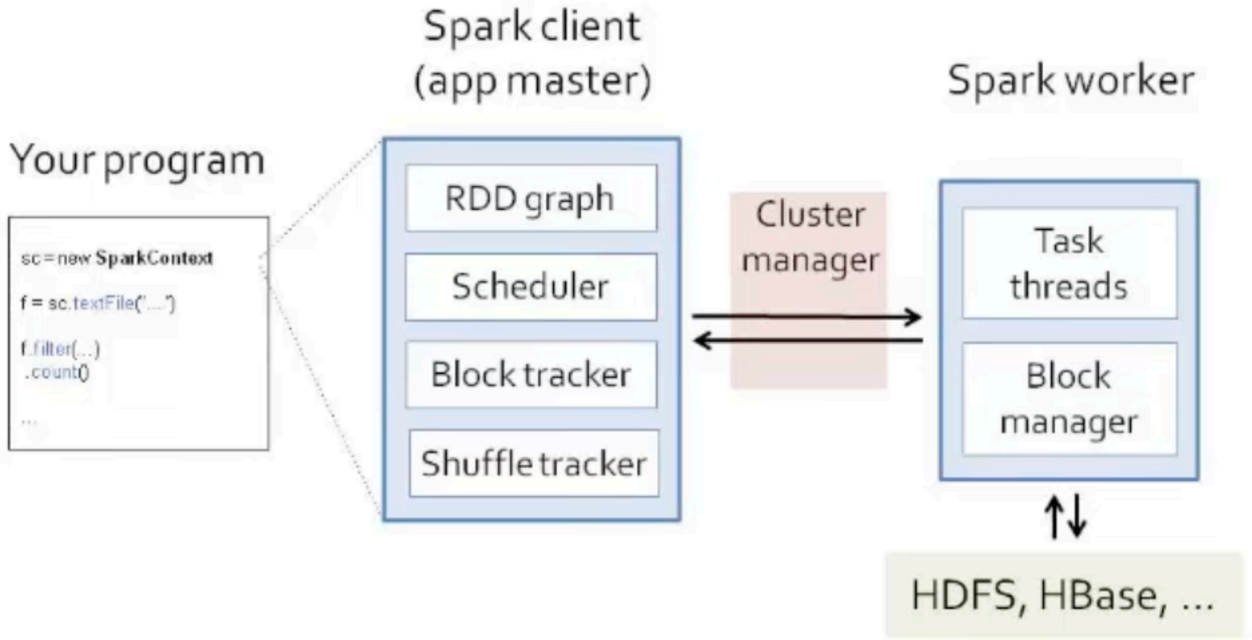
Example Job

```
val sc = new SparkContext(
  “spark://...”, “MyJob”, home, jars)

val file = sc.textFile(“hdfs://...”)

val errors = file.filter(_.contains(“ERROR”))

errors.cache()

errors.count()
```

Resilient distributed datasets (RDDs)

Action

Components



Example Job

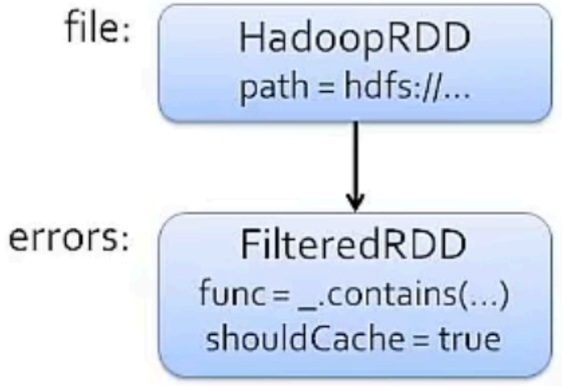
```
val sc = new SparkContext(
  "spark://...", "MyJob", home, jars)

val file = sc.textFile("hdfs://...")

val errors = file.filter(_.contains("ERROR"))

errors.cache()

errors.count()
```

Resilient distributed datasets (RDDs)

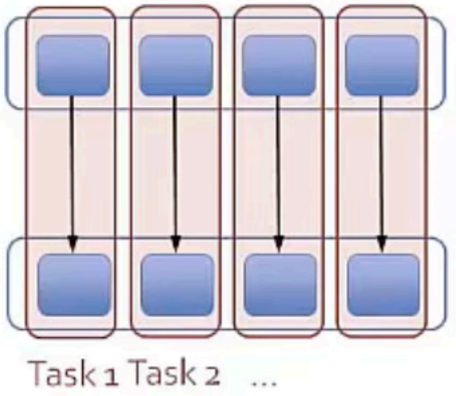
Action

RDD Graph

Dataset-level view:



Partition-level view:



RDD Interface

Set of *partitions* ("splits")

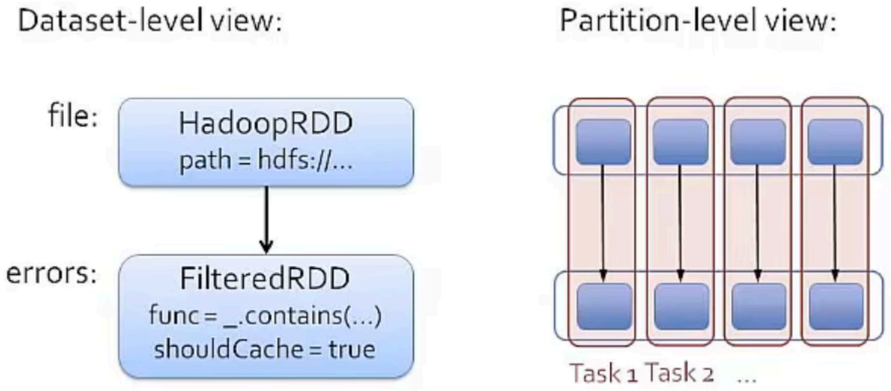
List of *dependencies* on parent RDDs

Function to *compute* a partition given parents

Optional *preferred locations*

Optional *partitioning info* (Partitioner)

RDD Graph



Example: FilteredRDD

partitions = same as parent RDD

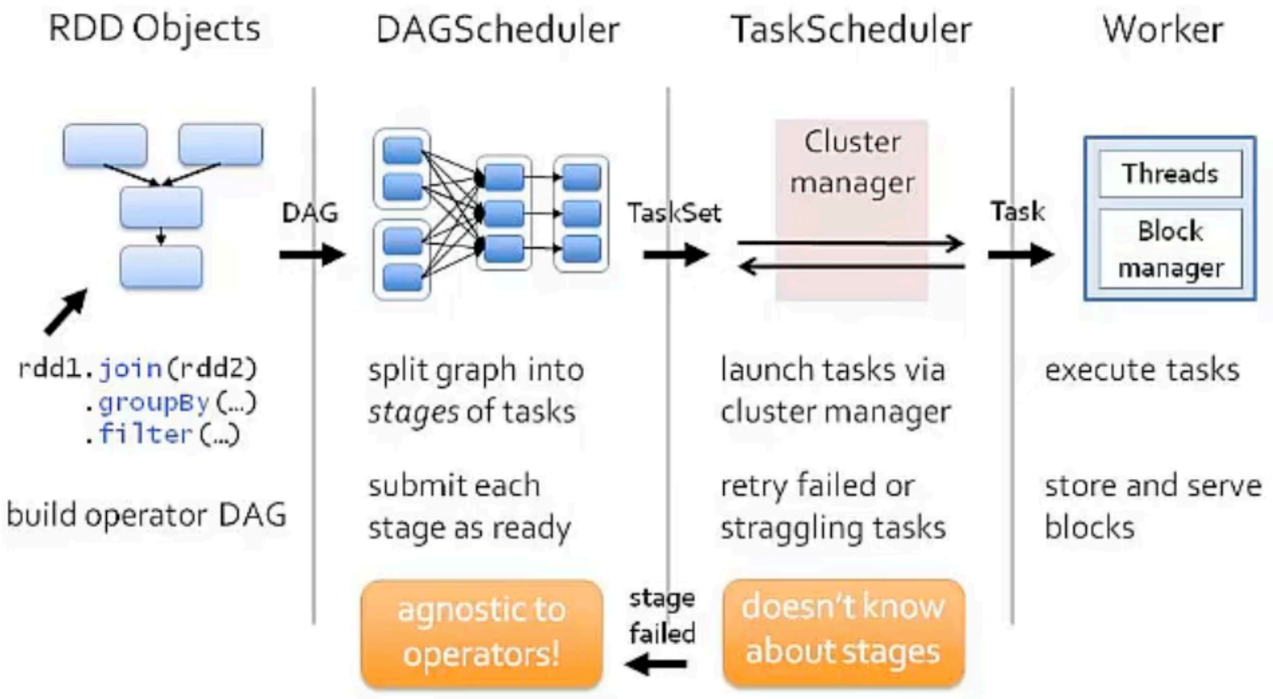
dependencies = "one-to-one" on parent

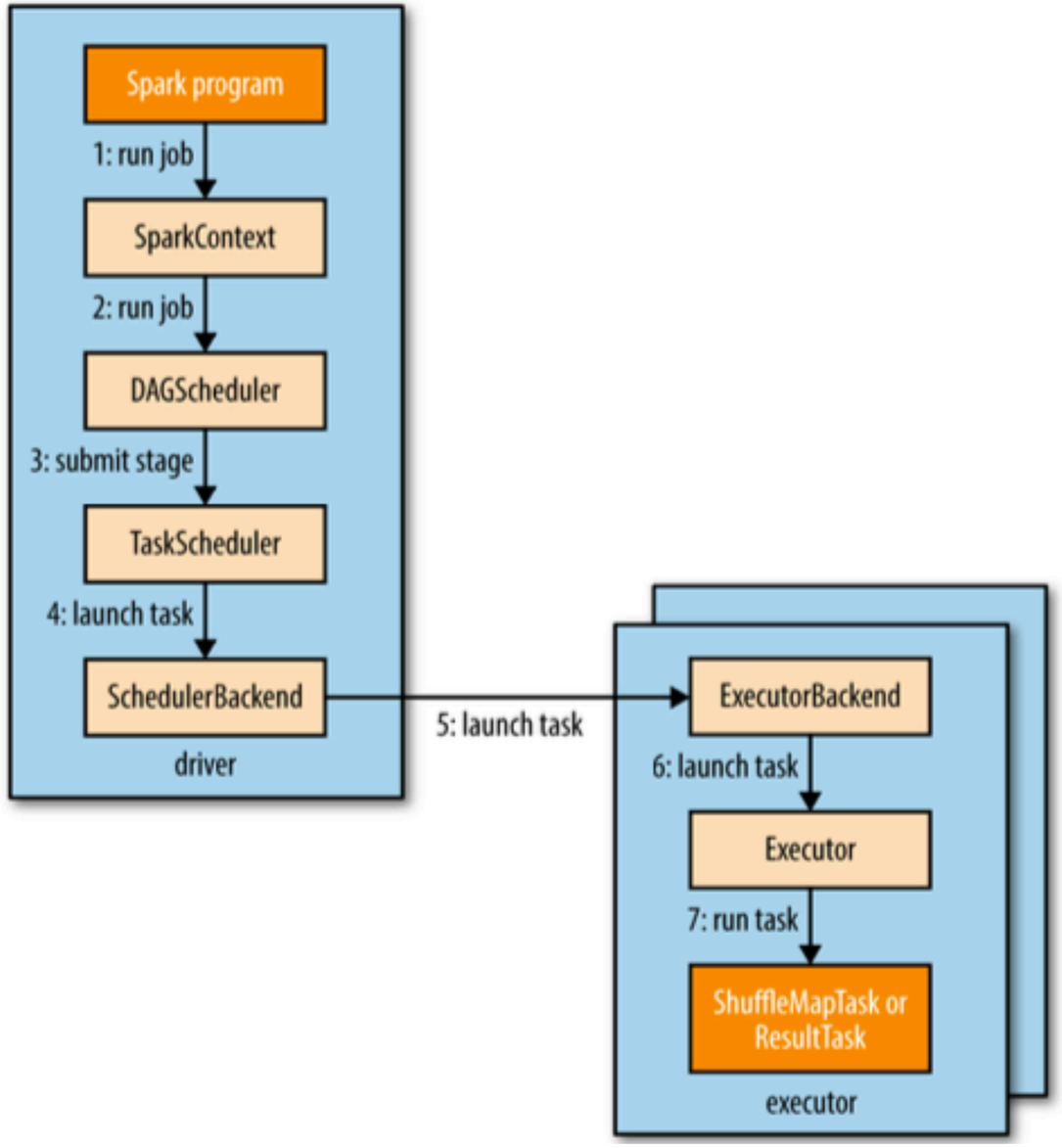
compute(*partition*) = compute parent and filter it

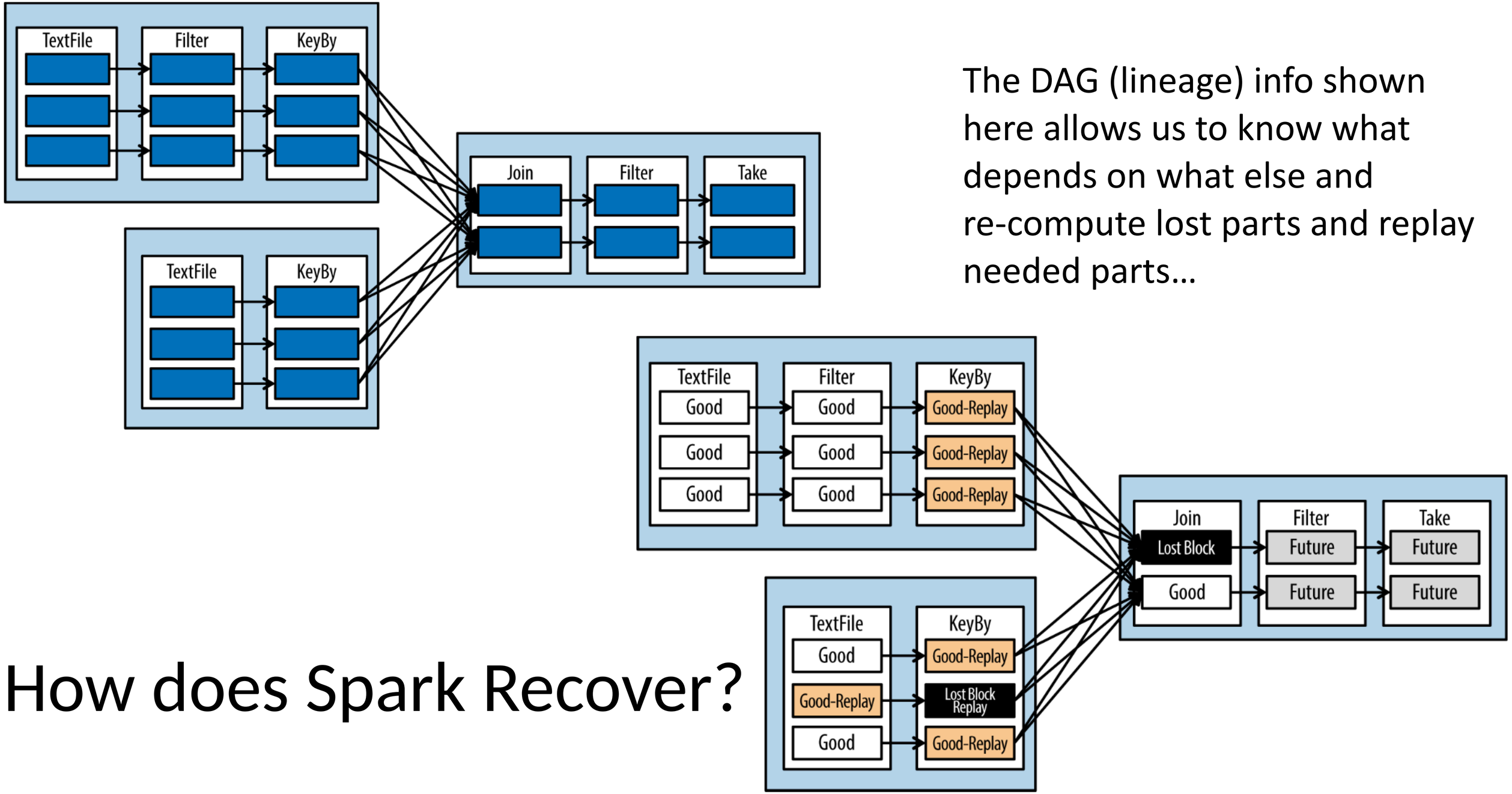
preferredLocations(*part*) = none (ask parent)

partitioner = none

Scheduling Process







The DAG (lineage) info shown here allows us to know what depends on what else and re-compute lost parts and replay needed parts…

How does Spark Recover?

Lazy Evaluation and Efficiency

Spark engages in pipelining and partition pruning, and more complex query optimization where needed

The latter is most useful with dataframes or SQL.

```
sc = SparkContext("local", "Lazy Evaluation Example")

# Create an RDD from a list
data = [1, 2, 3, 4, 5]
rdd = sc.parallelize(data)

# Define a transformation that would be costly (e.g., a map that involv
# For illustration, we'll use a simple operation, but imagine this bein
costly_transformation = rdd.map(lambda x: x * 2)

# Another transformation that filters out most of the data
# Let's say we end up not needing the costly computation for the final
filtered_rdd = costly_transformation.filter(lambda x: x > 10)  # This w

# An action that triggers the computation
result = filtered_rdd.collect()

# Print the result
print(result)
```

How Spark works: RDDs and DAGs

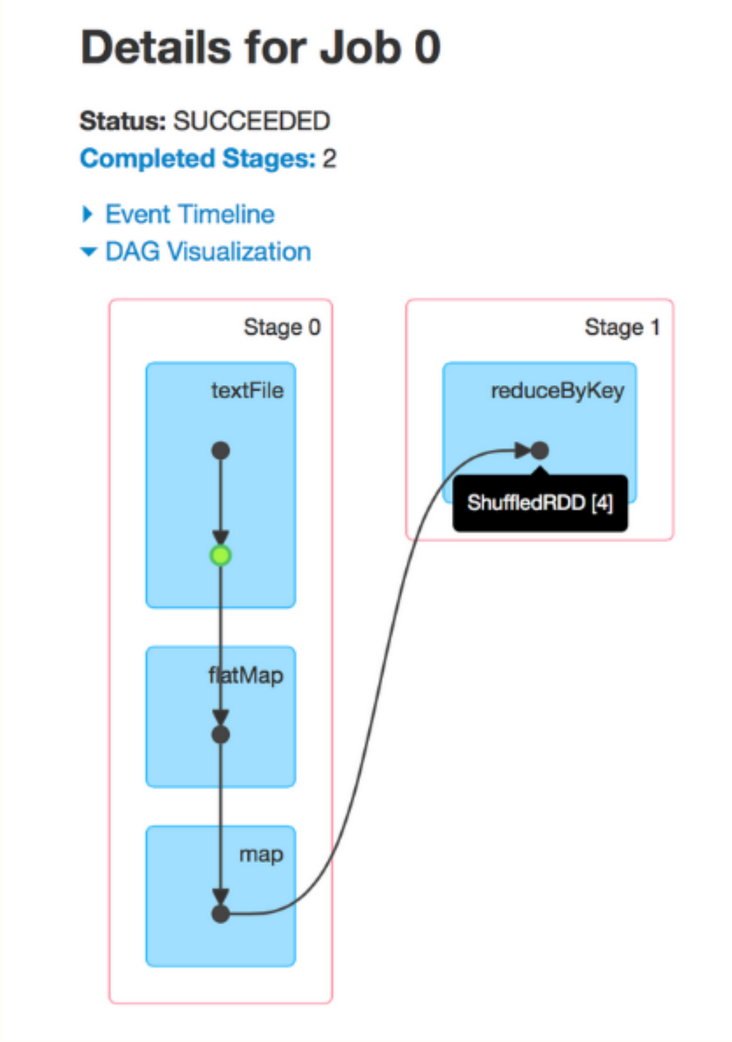
- Spark uses a Directed Acyclic Graph (DAG) of tasks. By organizing a program into this DAG it can decide where to schedule the tasks, and provide resiliency guarantees on these tasks

- Spark keeps large working datasets in memory between these tasks. To do this, Spark has a central abstraction, the Resilient Distributed Dataset (RDD): a read-only collection of objects partitioned over the cluster.

- One or more RDDs are loaded as input. Transformations create a set of target RDDs which may be saved or used to compute a result

How Spark Works (contd)

- Spark is Lazy. No computations are done until they are absolutely needed
- When the final action is done, Spark walks back on the DAG, figures the tasks(stages) to run, and the order in which they need doing, and starts running them.
- The job runs in the context of an application, represented by a SparkContext instance
- Because each RDD is immutable, a stage that fails can be run again by obtaining the starting RDD as the finishing RDD of the previous stage.



Business Impact

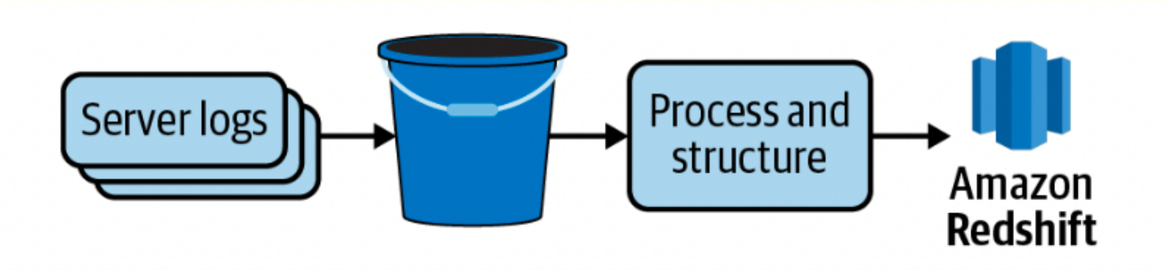
- Databricks was founded in 2013 by the founders of Spark from the AMPLab project
- It is now a cloud company with a Data Lake, AI, and Cloud offering.
- Spark is a part of the standard data engineering infrastructure and is used at almost every large company
- Microsoft and Databricks co-offer products on Azure
- On September 14 2023, Databricks raised 500 million putting its valuation at 43 Billion.

Modern Systems (sorta)

Hive: back to SQL

- Hive was created by Jeff Hammerbacher et. al at Facebook
- Hive was created so that analysts with good SQL skills but poor to non-existent java skills could make queries on the huge amount of data that facebook had
- Hive consists of a local client that converts your sql queries to mapreduce jobs that are run on a Hadoop cluster. It organizes data in HDFS into tables
- A key component, one that we will see later is the metastore which stores metadata such as table schemas
- Hive can also use Apache Tez and Apache Spark.

Lets revisit….



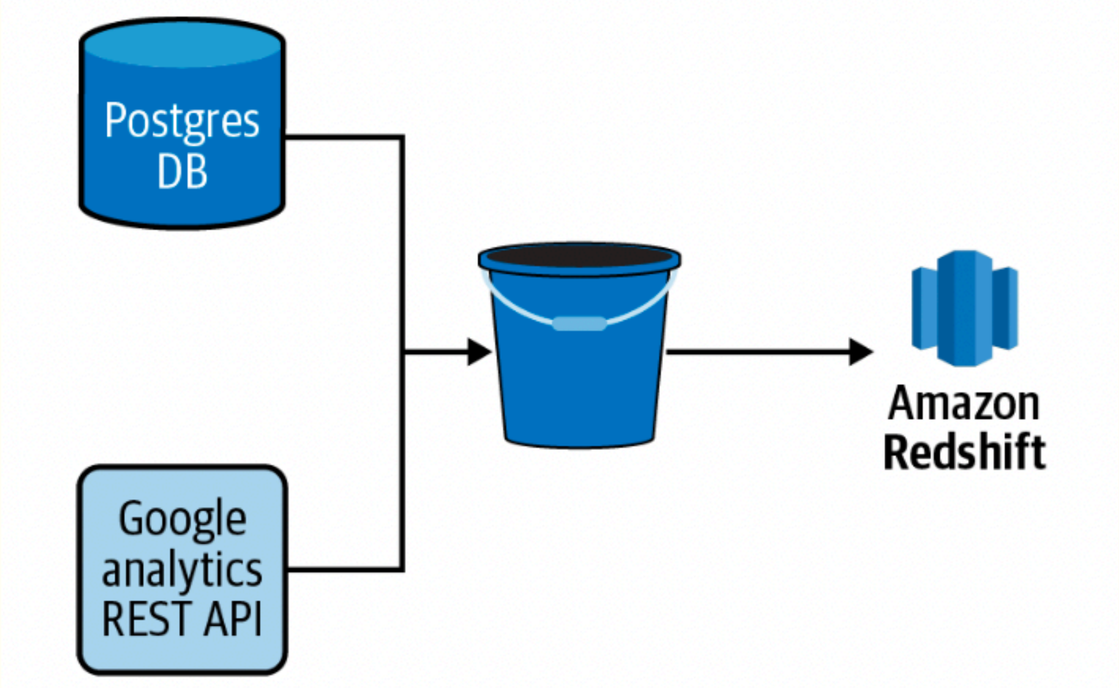
Where does data come from?
And where is it stored?

E-commerce databases

Logs

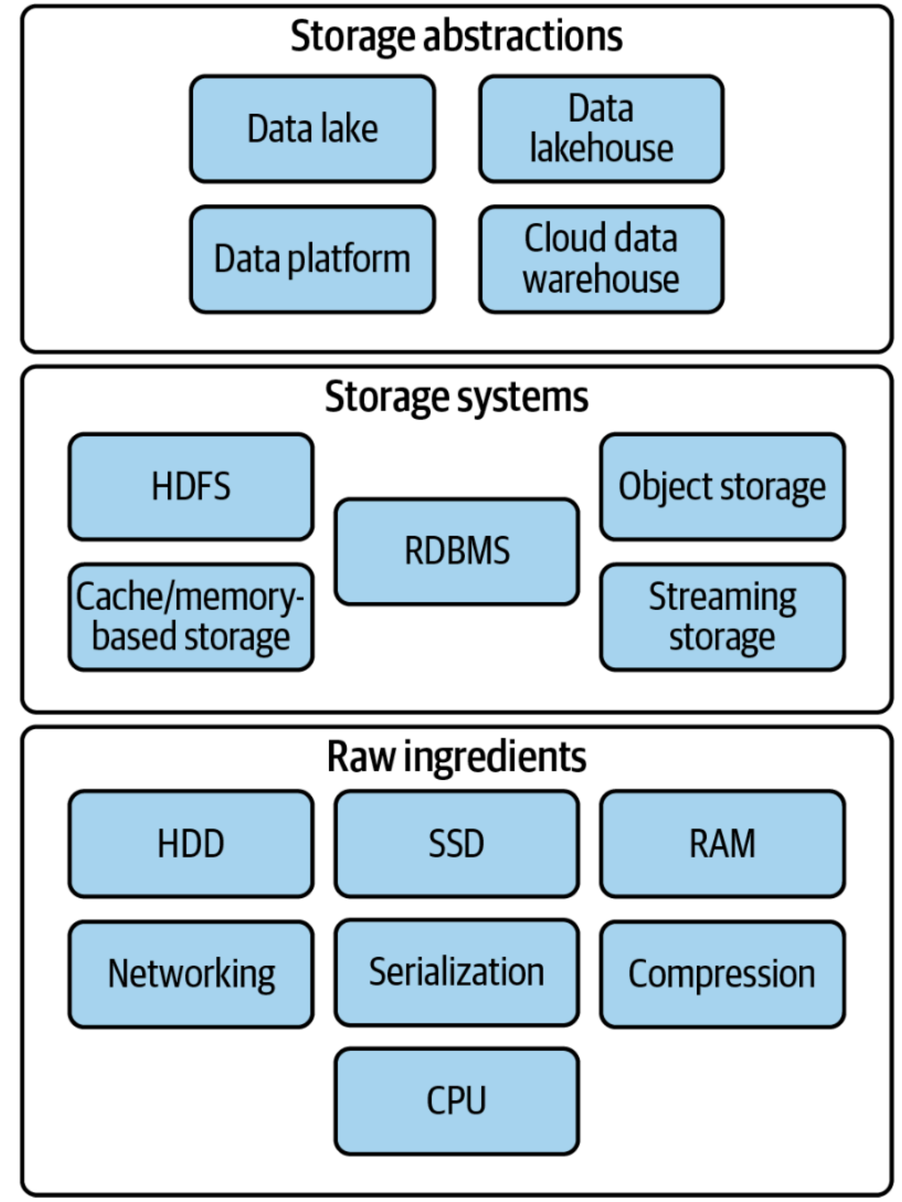
Customer tracking

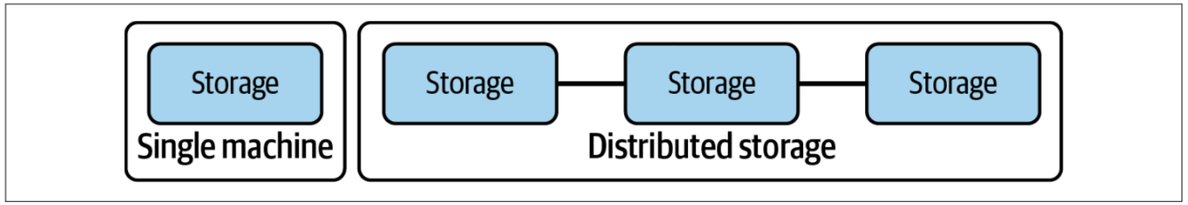
Other sites



Storage Abstractions

- We have evolved from building **data warehouses** on RDBMS to columnar databases on-premise to cloud storage
- **Data Lakes** were originally created for Hadoop/HDFS systems, and are now backed with object storage storing any kind of file, with HDFS clusters being spun up on demand.
- **Data lakehouse** combines the warehouse with the lake by adding SQL querying, schema support, and metadata management.
- **Data platforms** add management and catalog properties that allow us to create registries across all the data we have.



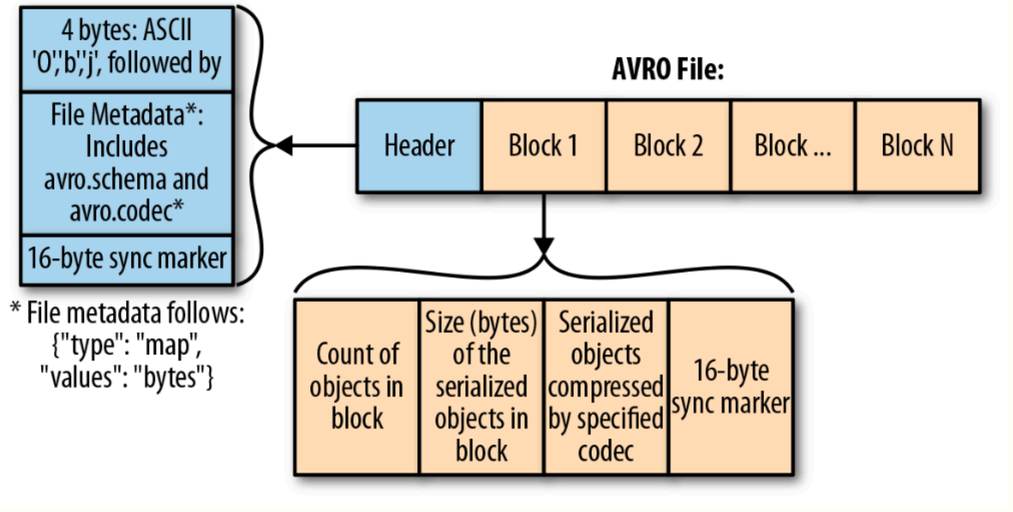


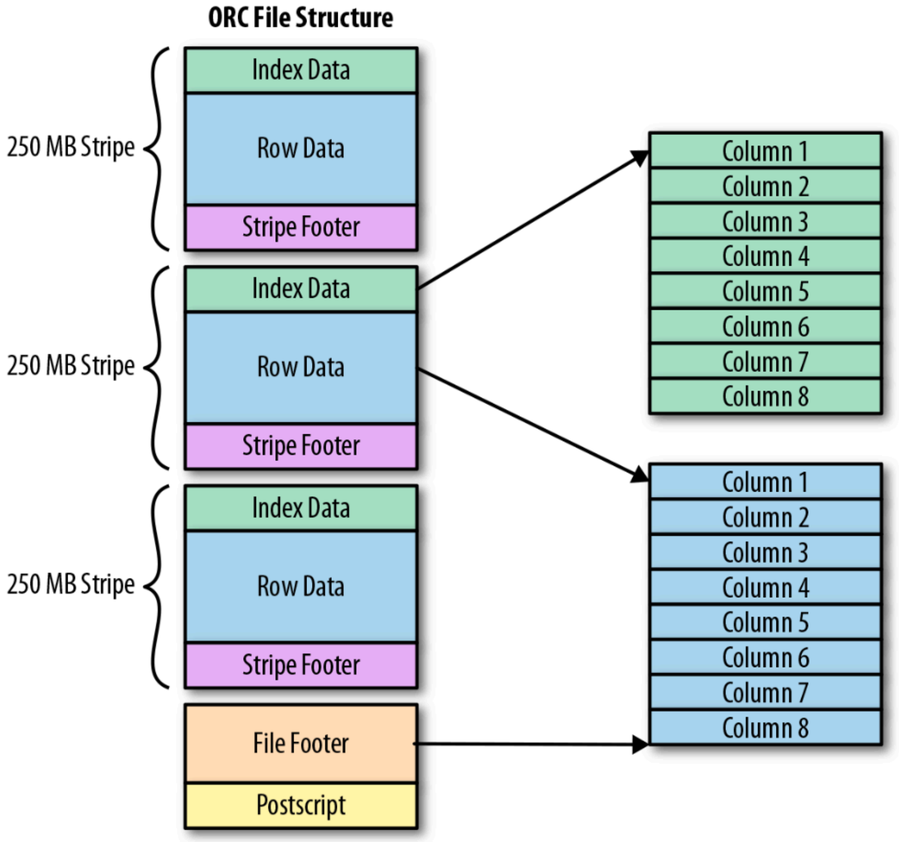
New Lake File Formats

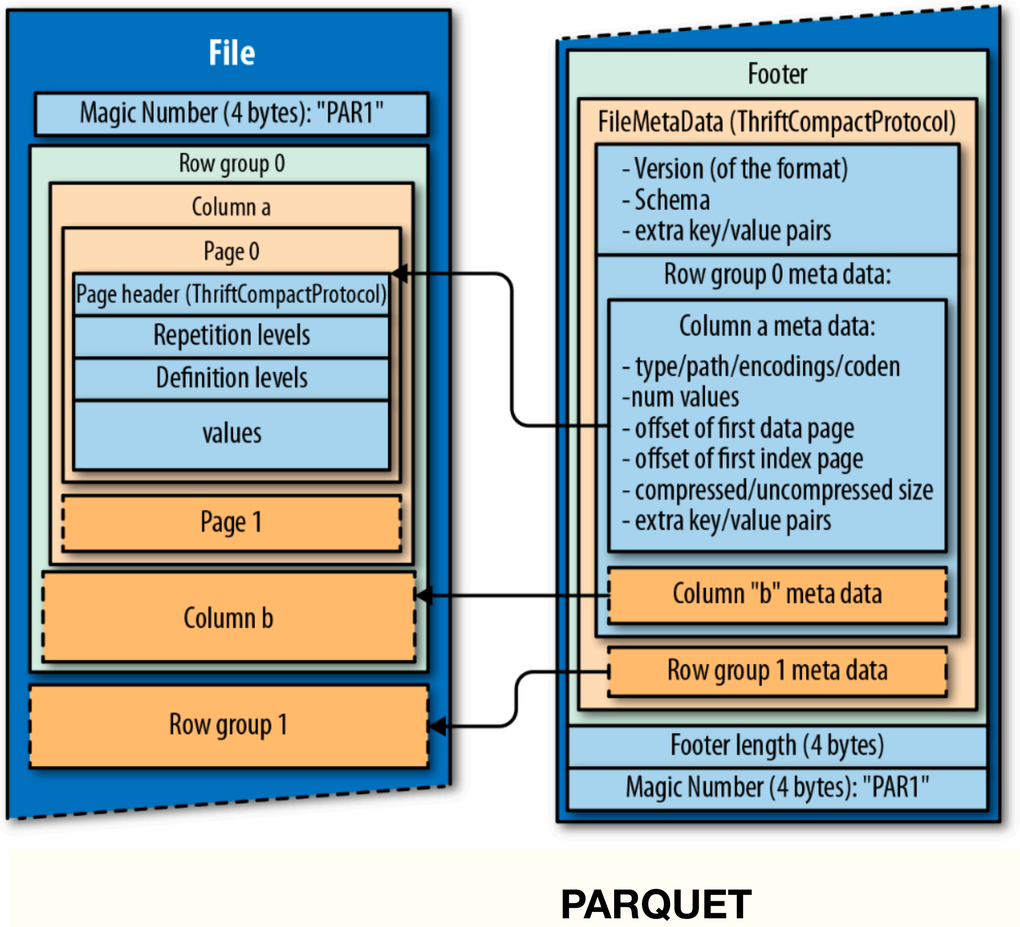
FORMAT	COLUMNAR	COMPRESSION	SUPPORT
AVRO	✗	GOOD	HADOOP SPARK ATHENA PRESTO
PARQUET	✓	GREAT	HADOOP SPARK ATHENA PRESTO
ORC	✓	EXCELLENT	HADOOP SPARK ATHENA PRESTO
CARBONDATA	✓	GOOD	HADOOP SPARK

	ORC	Parquet	Avro
Row or column	Column	Column	Row
Compression	Great	Great	Good
Speedup (compared to text file)	10–100x	10–100x	10x
Schema evolution	Good	Better	Best
Platforms	Hive, Spark, Presto	Hive, Spark, Presto	Hive, Spark
Splittability	Best	Best	Better
File statistics	Yes	Yes	No
Indexes	Yes	Yes	No
Bloom filters	Yes	No	No

	Parquet	AVRO
Format Type	Column-based	Row-based
Built For/with	Optimized column-wise compression and querying in a splittable file format designed for efficient Map-Reduce processing	Compact binary storage and exchange of records, with schema evolution and support for many different programming languages
Schema Storage	Column metadata stored at the end of the file (allows for fast, one-pass writing)	Stored in human-readable JSON format at the beginning of each message or file
Use Case	Quickly query all the values from a particular column across a very large dataset. For example, compute the average price of purchases over millions of purchase records	Share entire records between applications. For example, event data of in-app purchases for use by many downstream applications such as logging, auditing, and business analytics


4 bytes: ASCII 'O','b','j', followed by
File Metadata*: Includes avro.schema and avro.codec*
16-byte sync marker
* File metadata follows: {"type": "map", "values": "bytes"}
AVRO File:
Header
Block 1
Block 2
Block ...
Block N
Count of objects in block
Size (bytes) of the serialized objects in block
Serialized objects compressed by specified codec
16-byte sync marker


ORC File Structure
250 MB Stripe
Index Data
Row Data
Stripe Footer
250 MB Stripe
Index Data
Row Data
Stripe Footer
250 MB Stripe
Index Data
Row Data
Stripe Footer
File Footer
Postscript
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8
Column 1
Column 2
Column 3
Column 4
Column 5
Column 6
Column 7
Column 8


File
Magic Number (4 bytes): "PAR1"
Row group 0
Column a
Page 0
Page header (ThriftCompactProtocol)
Repetition levels
Definition levels
values
Page 1
Column b
Row group 1
Footer
FileMetaData (ThriftCompactProtocol)
- Version (of the format)
- Schema
- extra key/value pairs
Row group 0 meta data:
Column a meta data:
- type/path/encodings/coden
-num values
- offset of first data page
- offset of first index page
- compressed/uncompressed size
- extra key/value pairs
Column "b" meta data
Row group 1 meta data
Footer length (4 bytes)
Magic Number (4 bytes): "PAR1"

PARQUET

Warehouse, or Lake, or Lakehouse or Platform?

	Processing data in the data lake (Spark)	Processing data in the data warehouse (SQL)
Flexibility	Processing done in the data lake brings additional flexibility because outputs can be used not just for data served in the data warehouse but also for data that can be delivered to or consumed by other users and/or systems.	Outputs of data processing are typically restricted to use in the data warehouse.
Developer productivity	Once trained, developers will appreciate the power and flexibility of Spark with its sophisticated testing frameworks and libraries to accelerate code delivery.	While not designed as a programming language, SQL's popularity means that finding people who know it is relatively easy, so using SQL instead of learning Spark can mean a shorter time to value.

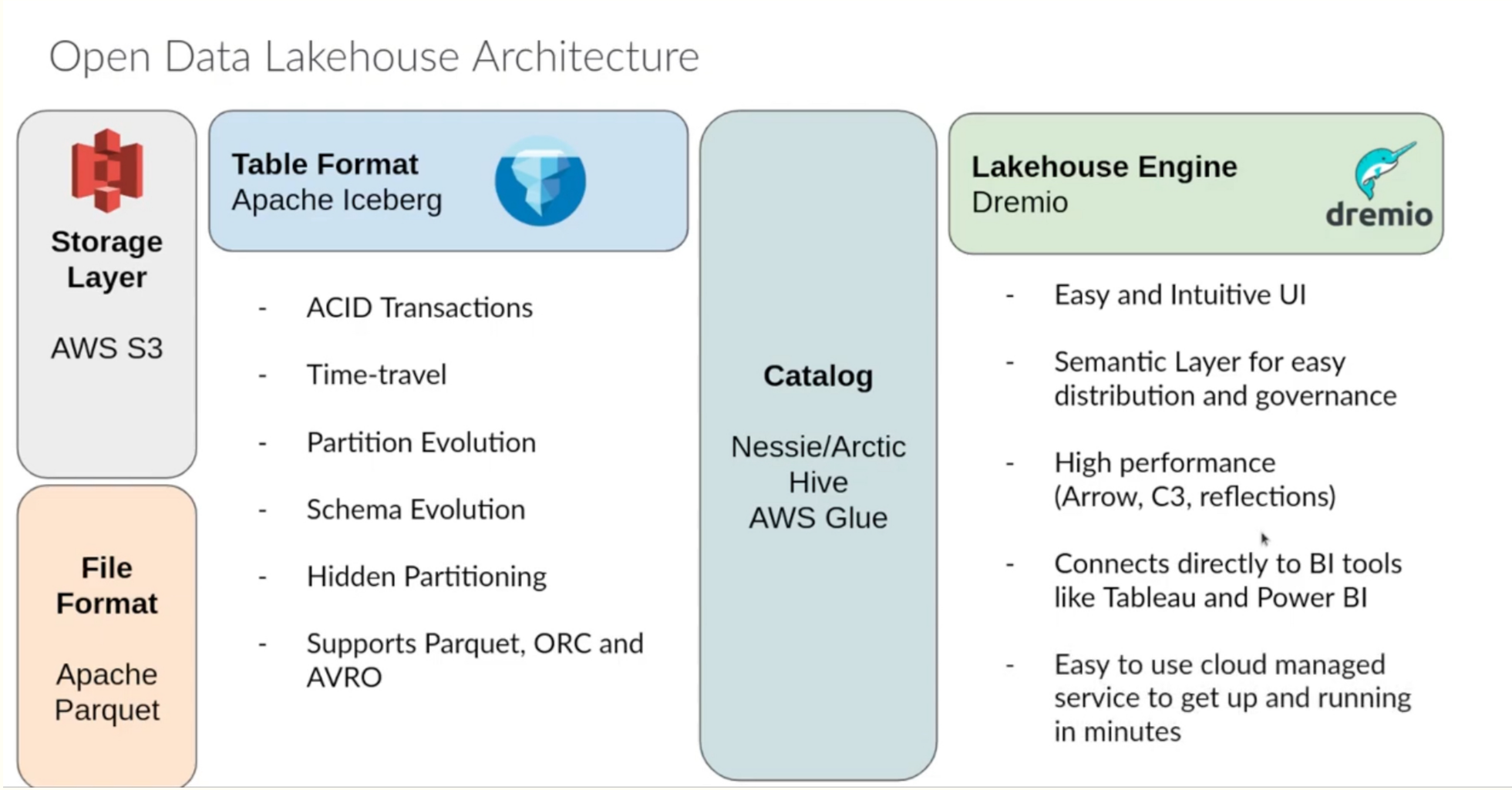
- Warehouse has a columnar database
- Lake has parquet and other files on object storage
- Lakehouse adds a table structure to the lake
- Platform adds warehouse/lake/etc

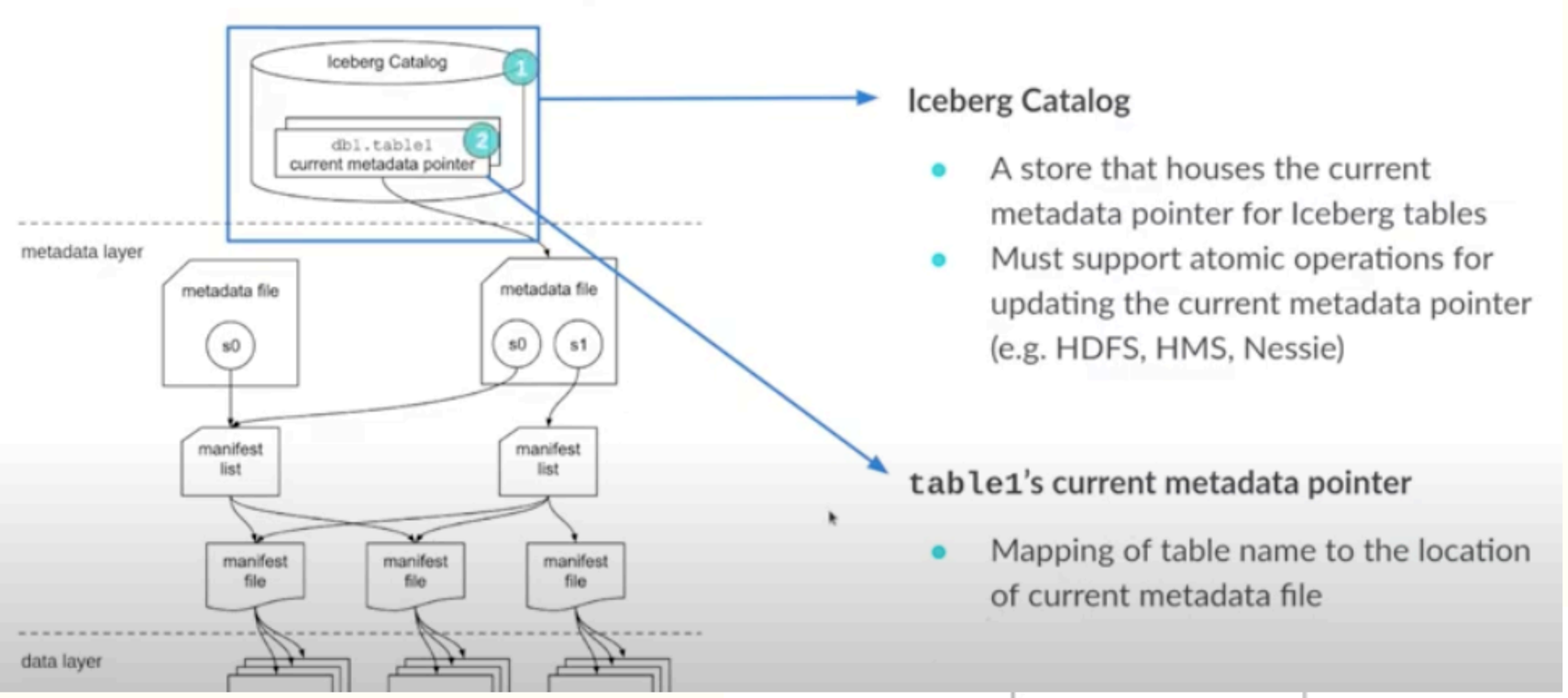
Data warehouse–only design	Data platform design
You only have a relational data source.	You have multiple data sources with structured and semistructured data.
You have control over your source data and a process in place to manage schema changes.	Your want to ingest and use data from multiple data sources, i.e., spreadsheets or SaaS products over which you don't have full control.
Your use cases are constrained to BI reports and interactive SQL queries.	You want to be able to use your data for machine learning and data science use cases in addition to traditional BI and data analytics.
You have a limited community of data users.	You have more and more users in your organization requiring access to the data to do their job.
Your data volumes are small enough to justify the cost of storing and processing all data inside a cloud data warehouse.	You want to optimize your cloud costs by using different cloud services for storing and processing data.

Really Modern Systems

Structured Lakehouses

An example of a Lakehouse





Iceberg Catalog

- A store that houses the current metadata pointer for Iceberg tables
- Must support atomic operations for updating the current metadata pointer (e.g. HDFS, HMS, Nessie)

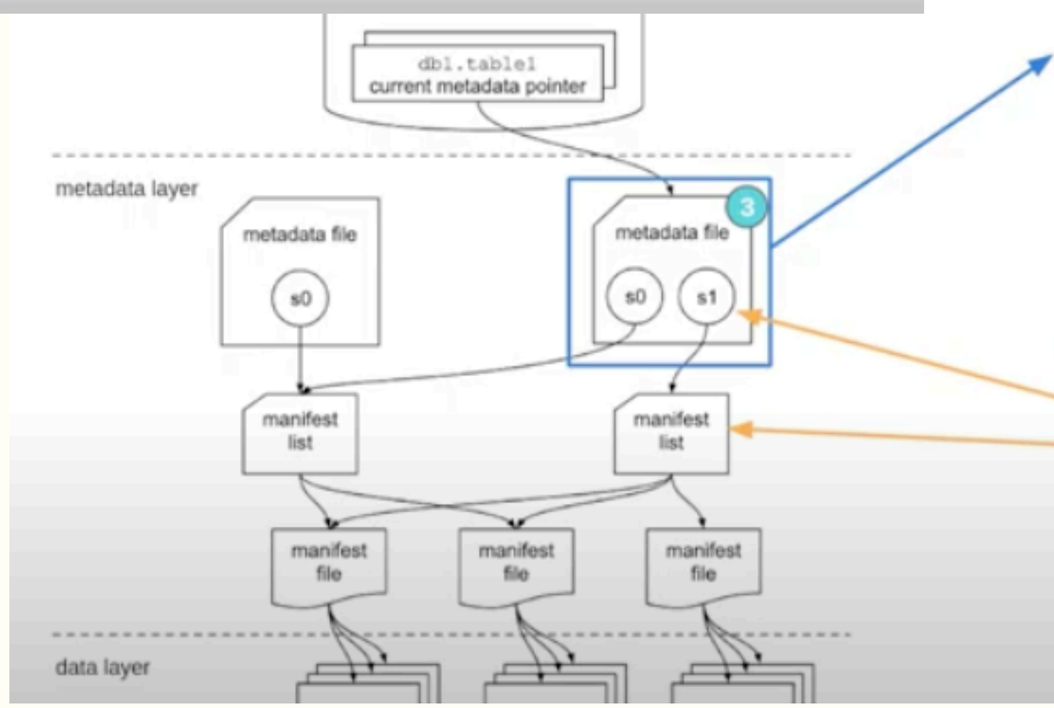
`table1`'s current metadata pointer

- Mapping of table name to the location of current metadata file

Components of a Lakehouse

- Catalog: an external database, or a hive metastore
- in some cases provides multi-process transactional guarantees
- keeps a pointer to the current metadata file

- metadata is arranged on a per snapshot basis
- consists of location and schema information, and snapshots
- keeps a pointer to the current snapshot

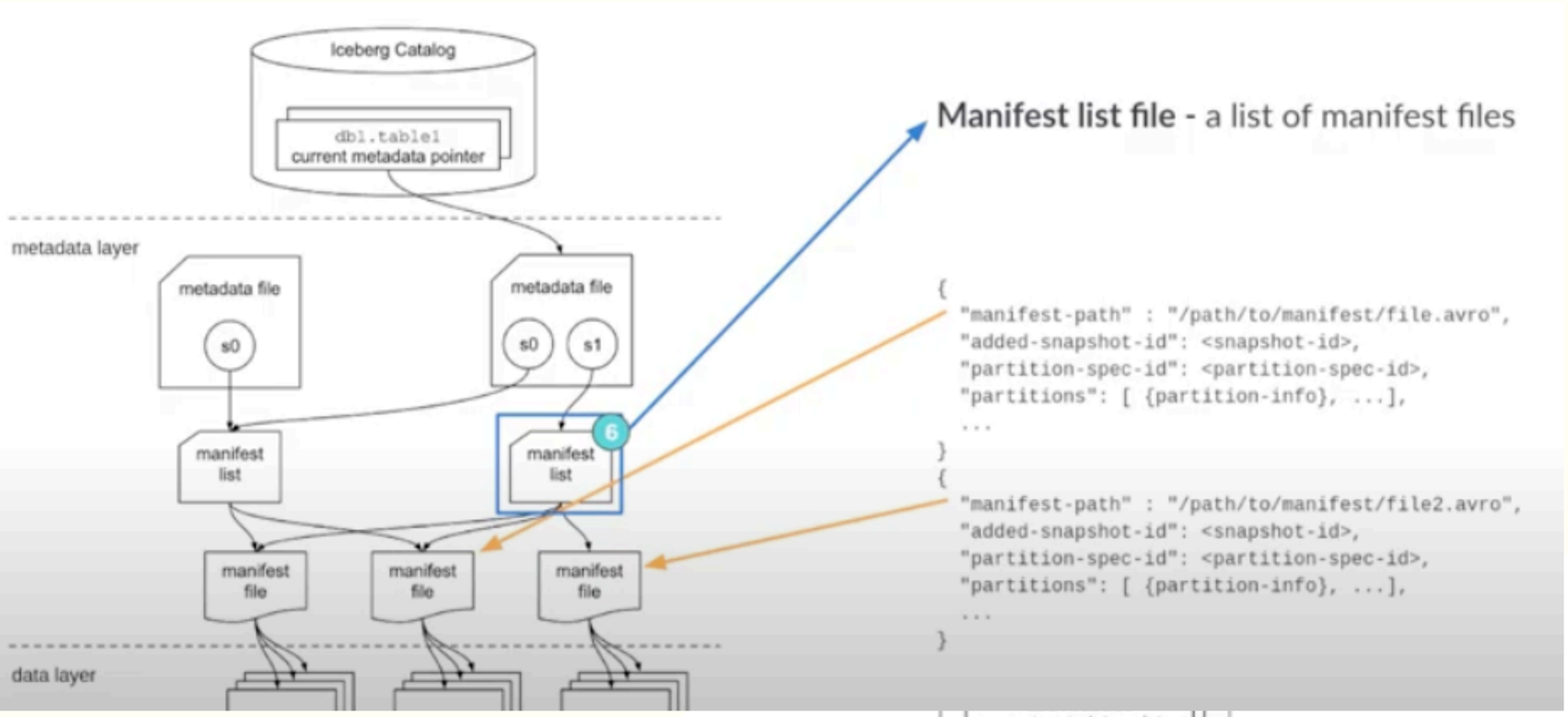


Metadata file - stores metadata about a table at a certain point in time

```
{
  "table-uuid" : "<uuid>",
  "location" : "/path/to/table/dir",
  "schema": {...},
  "partition-spec": [ {<partition-details>}, ...],
4 "current-snapshot-id": <snapshot-id>,
  "snapshots": [ {
      "snapshot-id": <snapshot-id>
5     "manifest-list": "/path/to/manifest/list.avro"
  }, ...],
  ...
}
```

Manifests

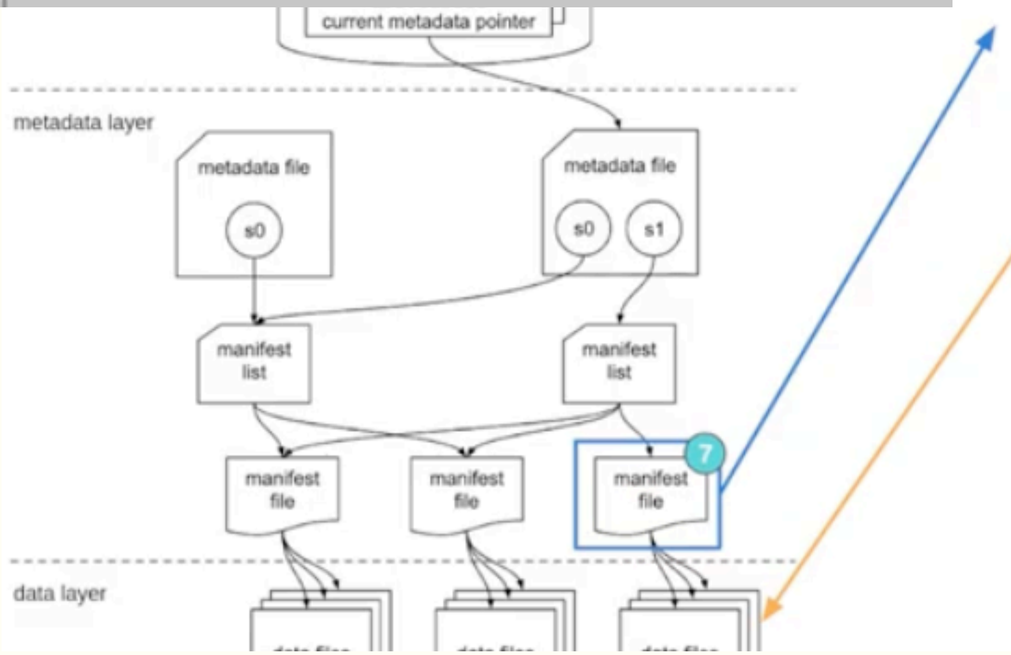
- metadata file points to a manifest list file
- the manifest list file has a set of manifest files



Manifest list file - a list of manifest files

```
{
  "manifest-path" : "/path/to/manifest/file.avro",
  "added-snapshot-id": <snapshot-id>,
  "partition-spec-id": <partition-spec-id>,
  "partitions": [ {partition-info}, ...],
  ...
}
{
  "manifest-path" : "/path/to/manifest/file2.avro",
  "added-snapshot-id": <snapshot-id>,
  "partition-spec-id": <partition-spec-id>,
  "partitions": [ {partition-info}, ...],
  ...
}
```

- each manifest file has a list of parquet, orc or avro files
- the manifest file also keeps statistics about the columns in the schema



Manifest file - a list of data files, along with details and stats about each data file

```
{
  "data-file": {
    "file-path": "/path/to/data/file.parquet",
    "file-format": "PARQUET",
    "partition": {"<part-field>":{"<data-type>":<value>}},
    "record-count": <num-records>,
    "null-value-counts": [{
      "column-index": "1", "value": 4
    }, ...],
    "lower-bounds": [{
      "column-index": "1", "value": "aaa"
    }, ...],
    "upper-bounds": [{
      "column-index": "1", "value": "eee"
    }, ...],
  }
  ...
}
```