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

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.*

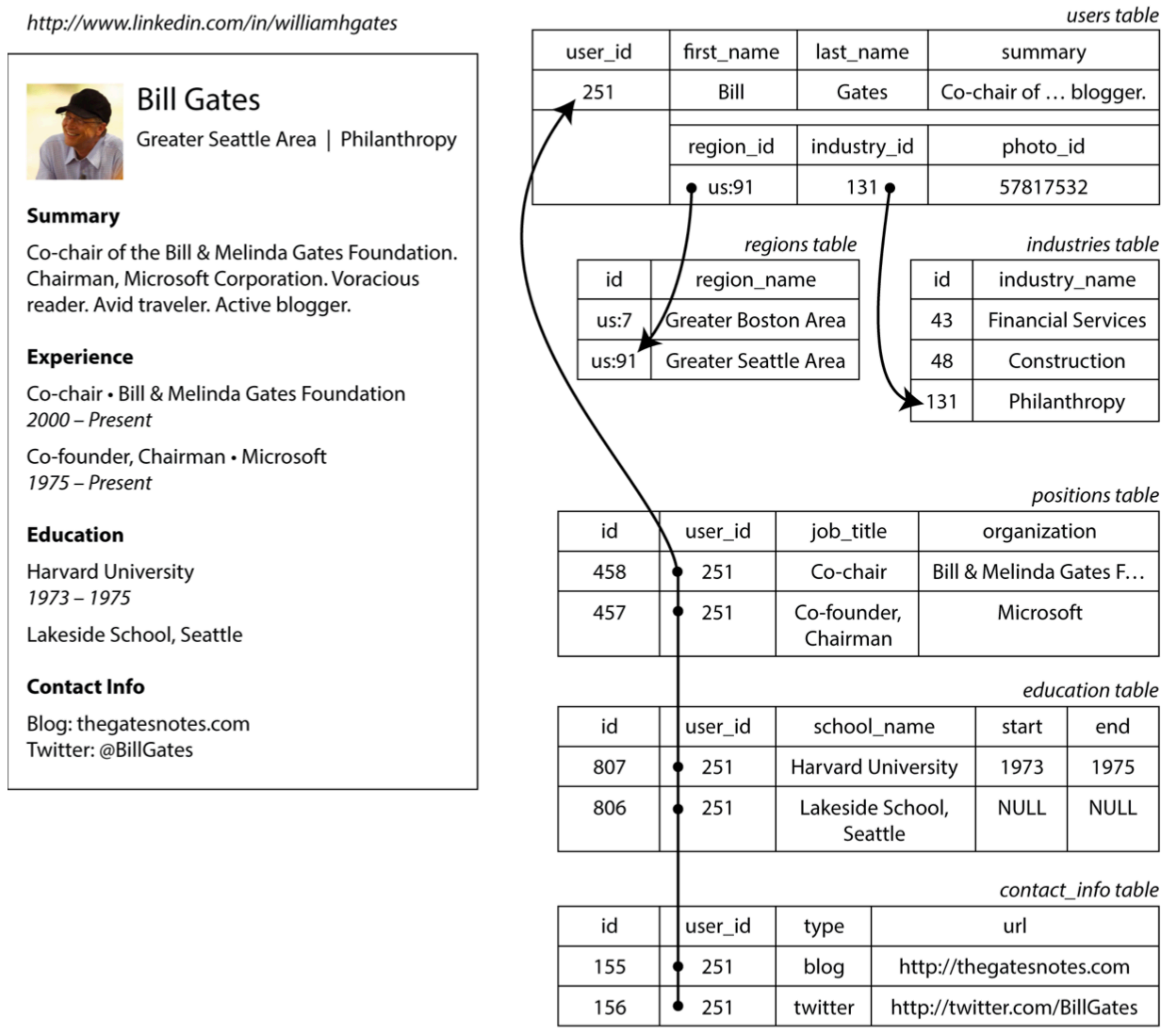


Table: contributors

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

```
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

```
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;
```

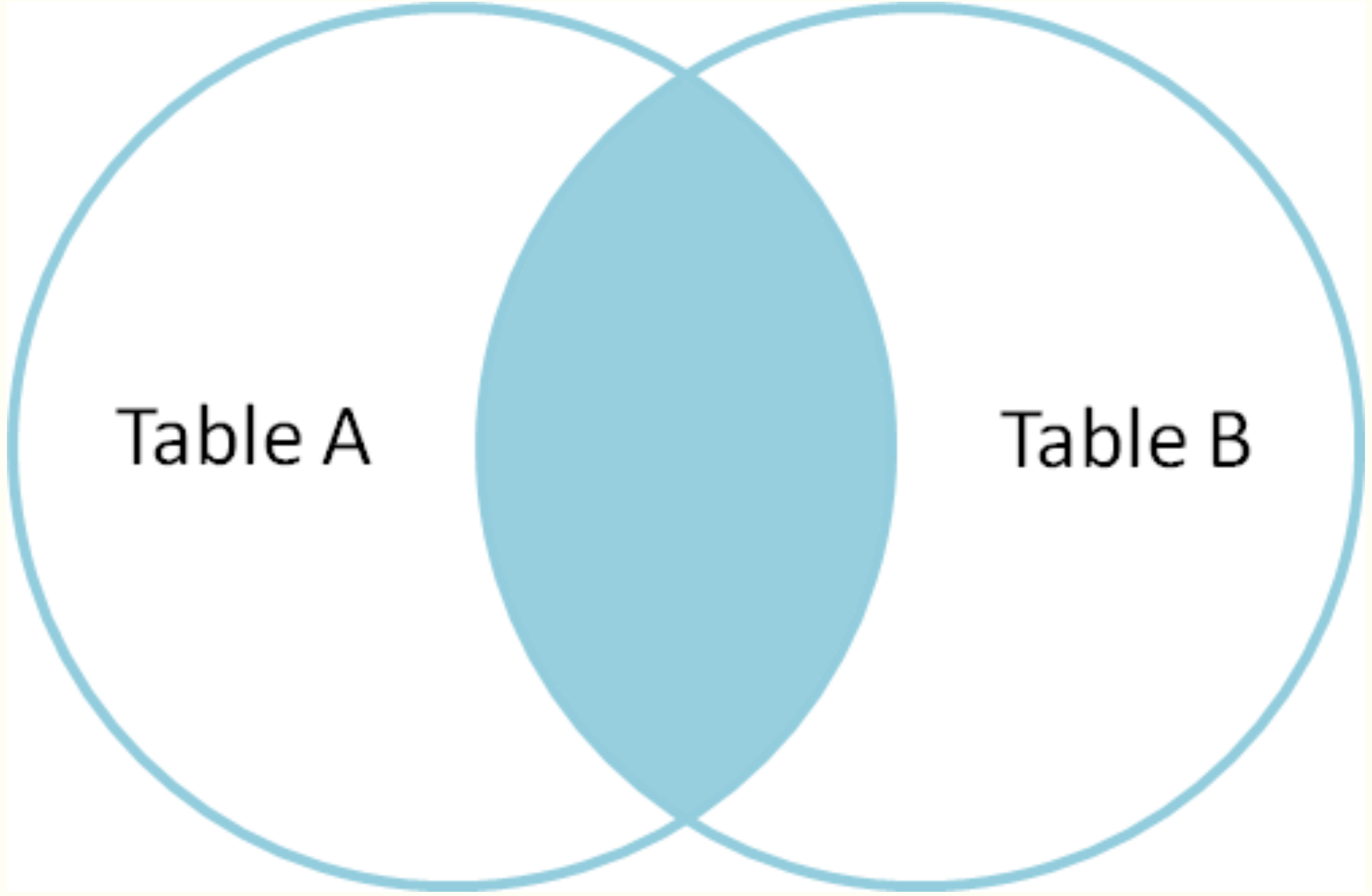
inserts and alters

```
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

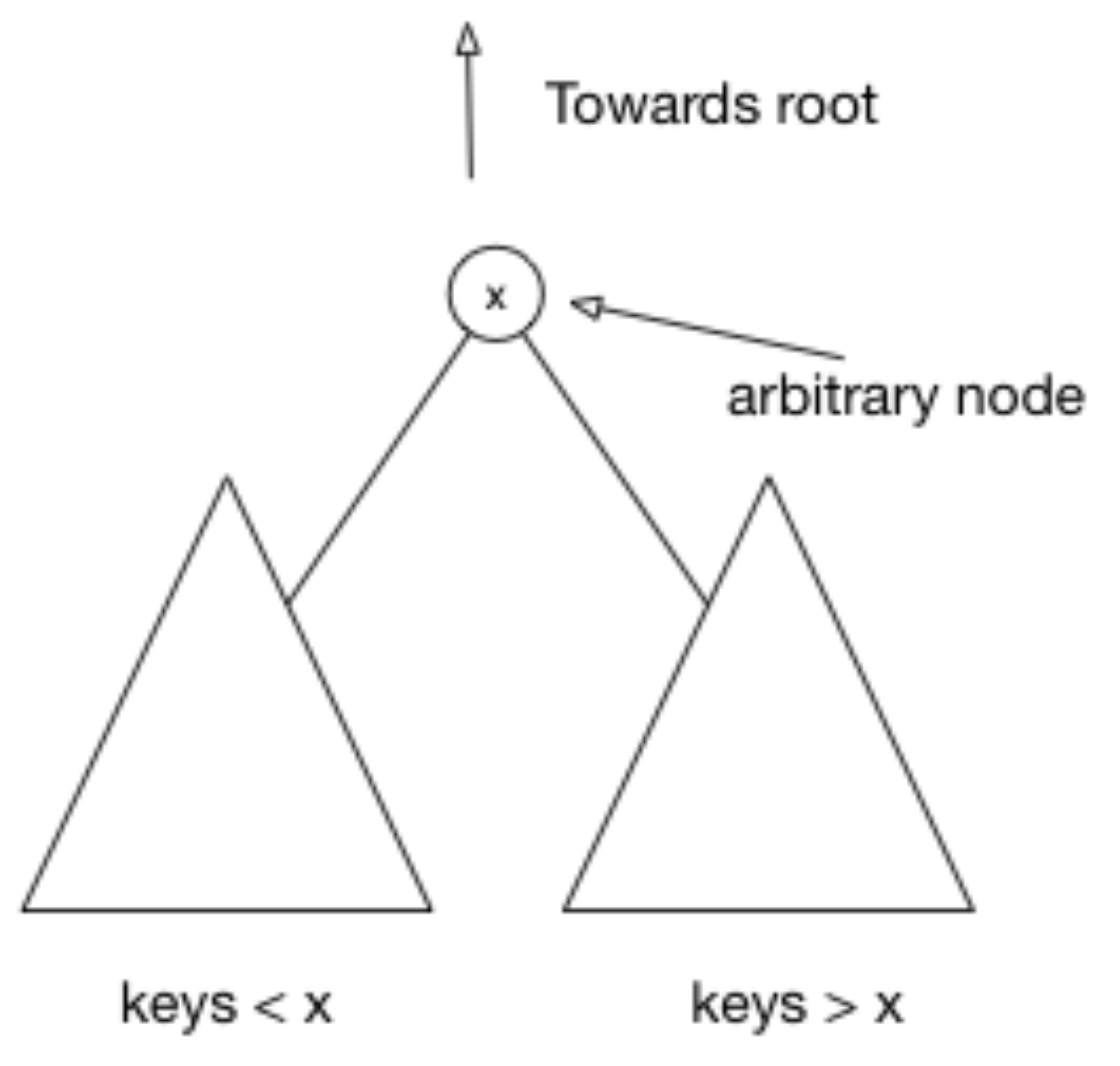
	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2

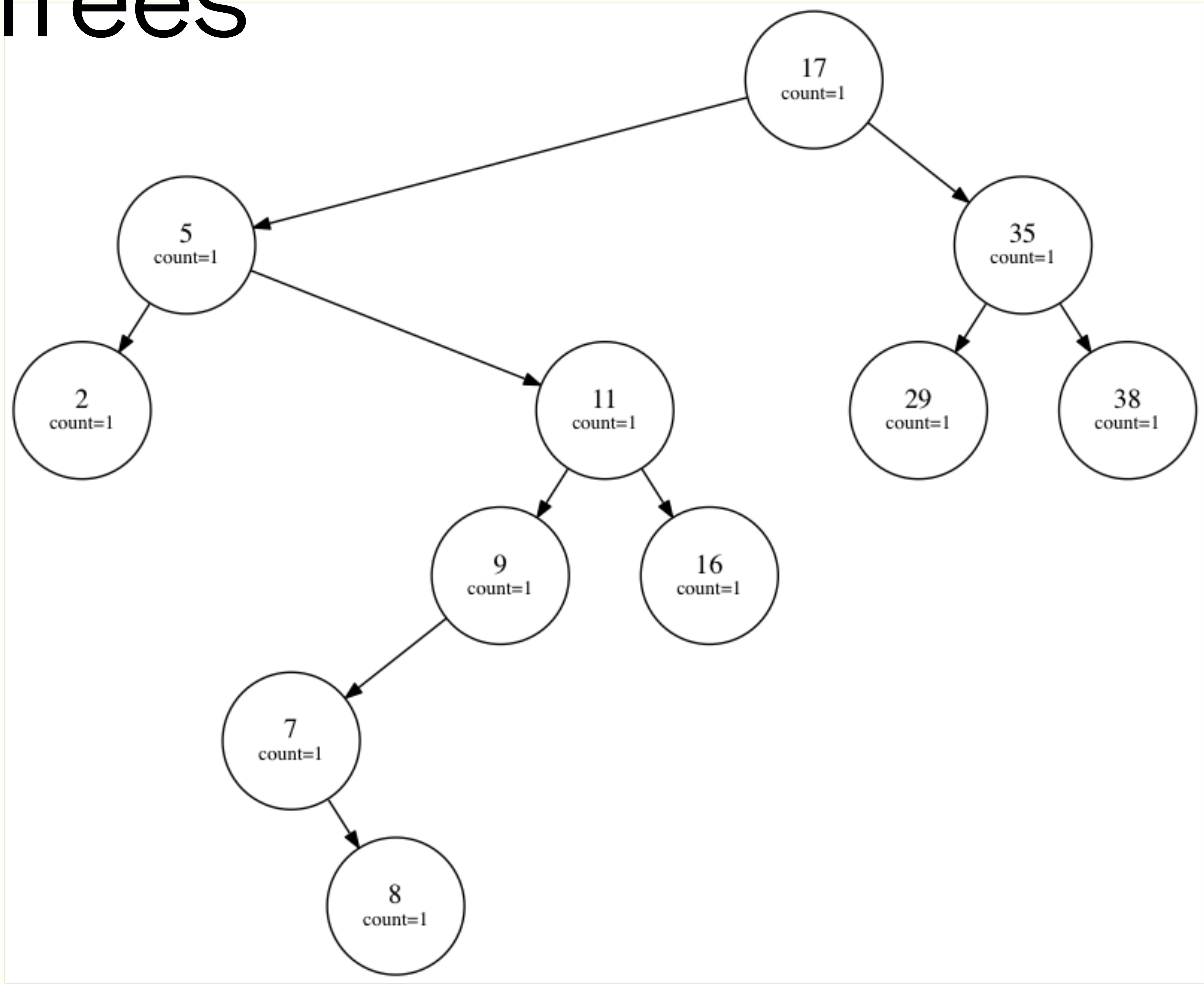
Storage Components of a RDBMS

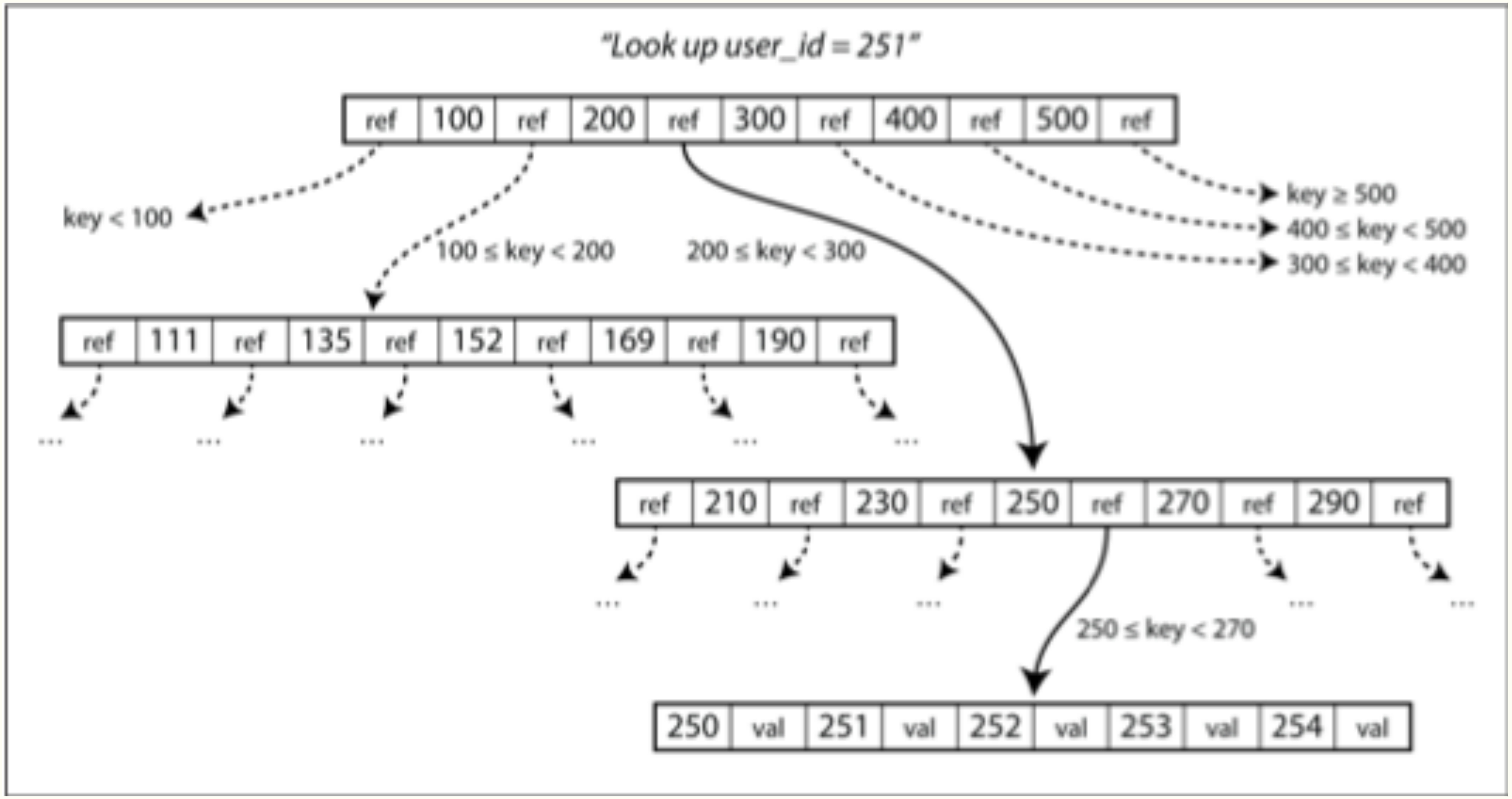
- the heap file: this is where the rows or columns are stored
- regular relational databases use row oriented heap files
- an index file(s): this is where the index for a particular attribute is stored
- sometimes you have a clustered index (all data stored in index) or covering index (some data is stored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

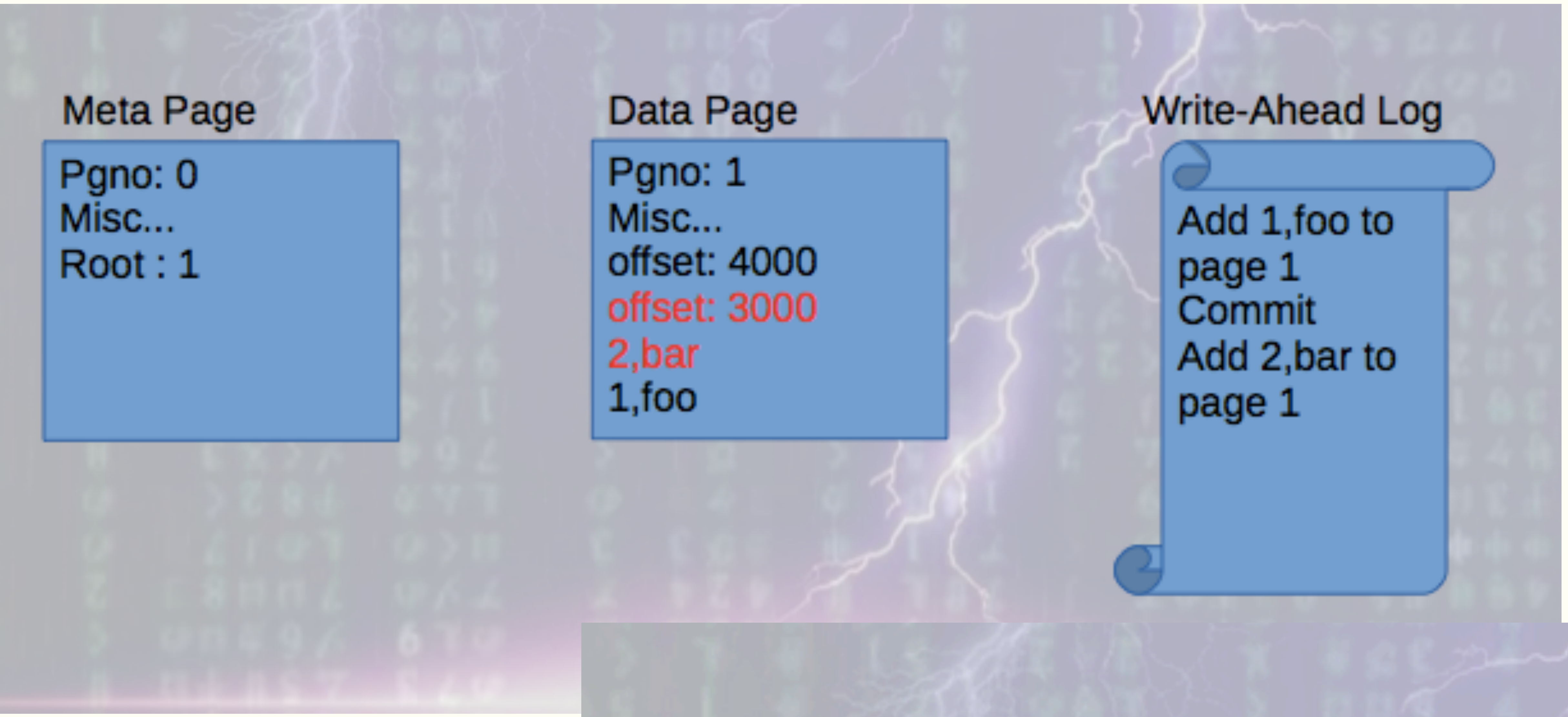


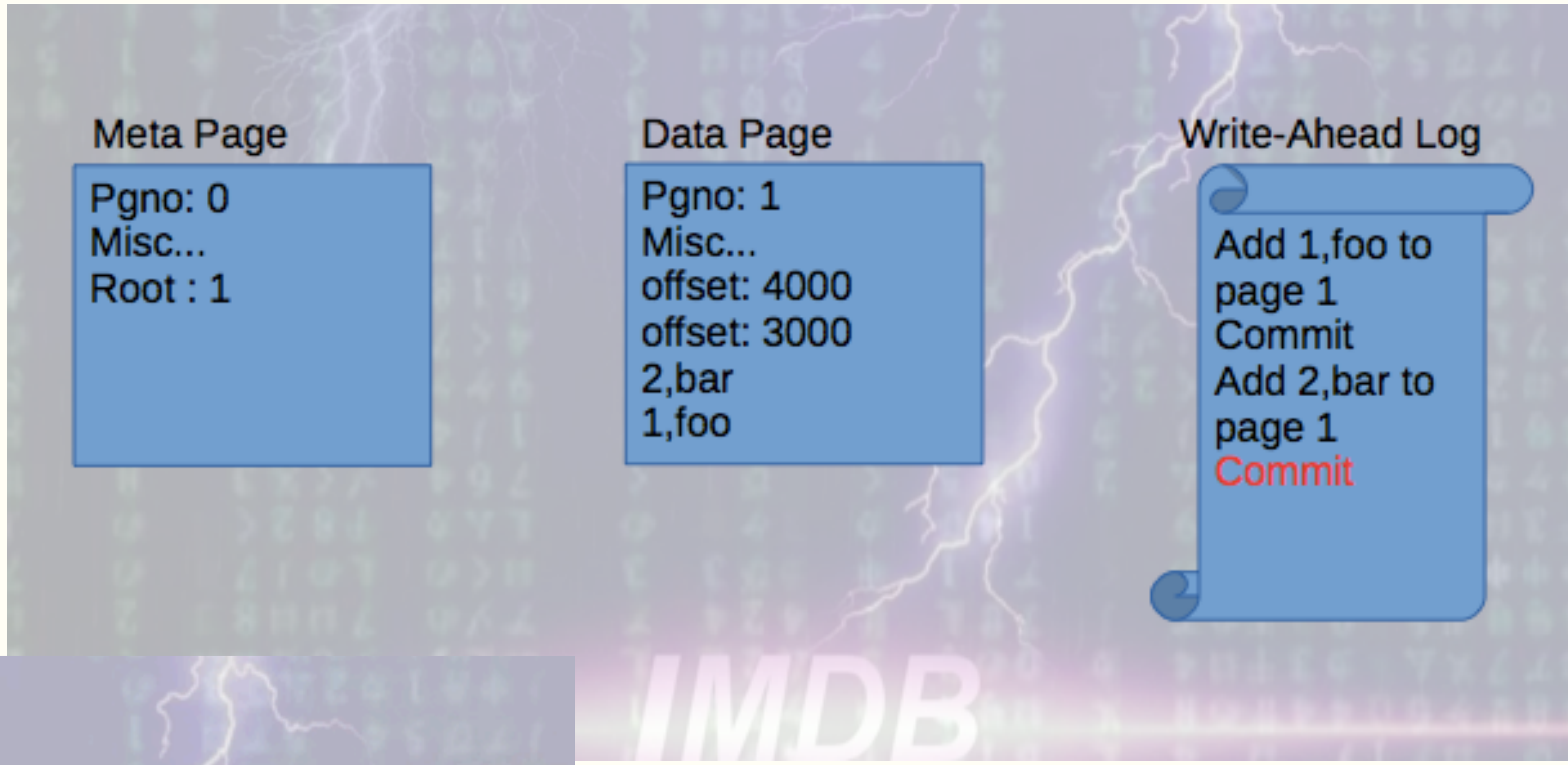


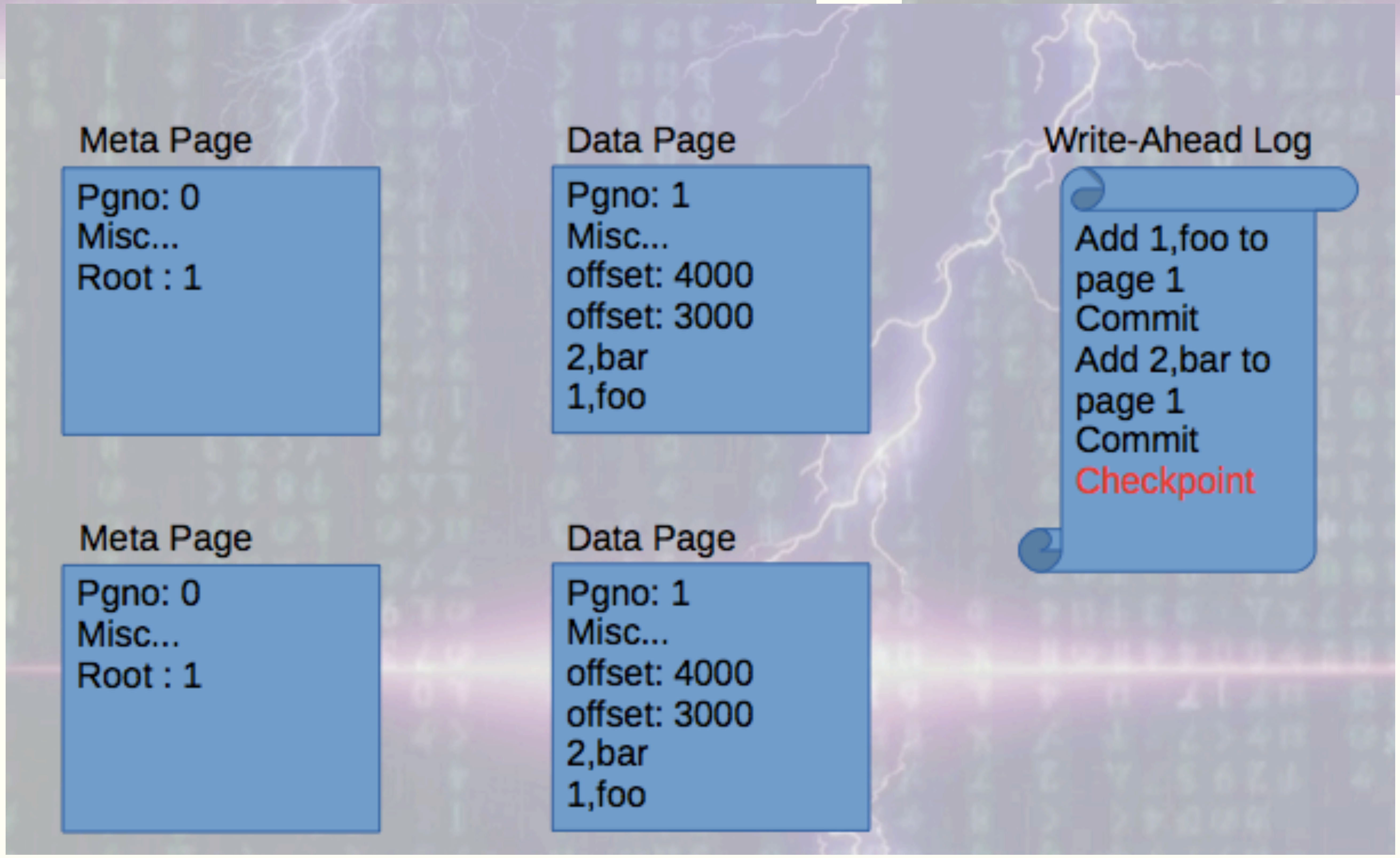

"Look up user_id = 251"
ref
100
ref
200
ref
300
ref
400
ref
500
ref
key < 100
100 ≤ key < 200
200 ≤ key < 300
key ≥ 500
400 ≤ key < 500
300 ≤ key < 400
ref
111
ref
135
ref
152
ref
169
ref
190
ref
ref
210
ref
230
ref
250
ref
270
ref
290
ref
250 ≤ key < 270
250
val
251
val
252
val
253
val
254
val

Btrees

WAL


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

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.

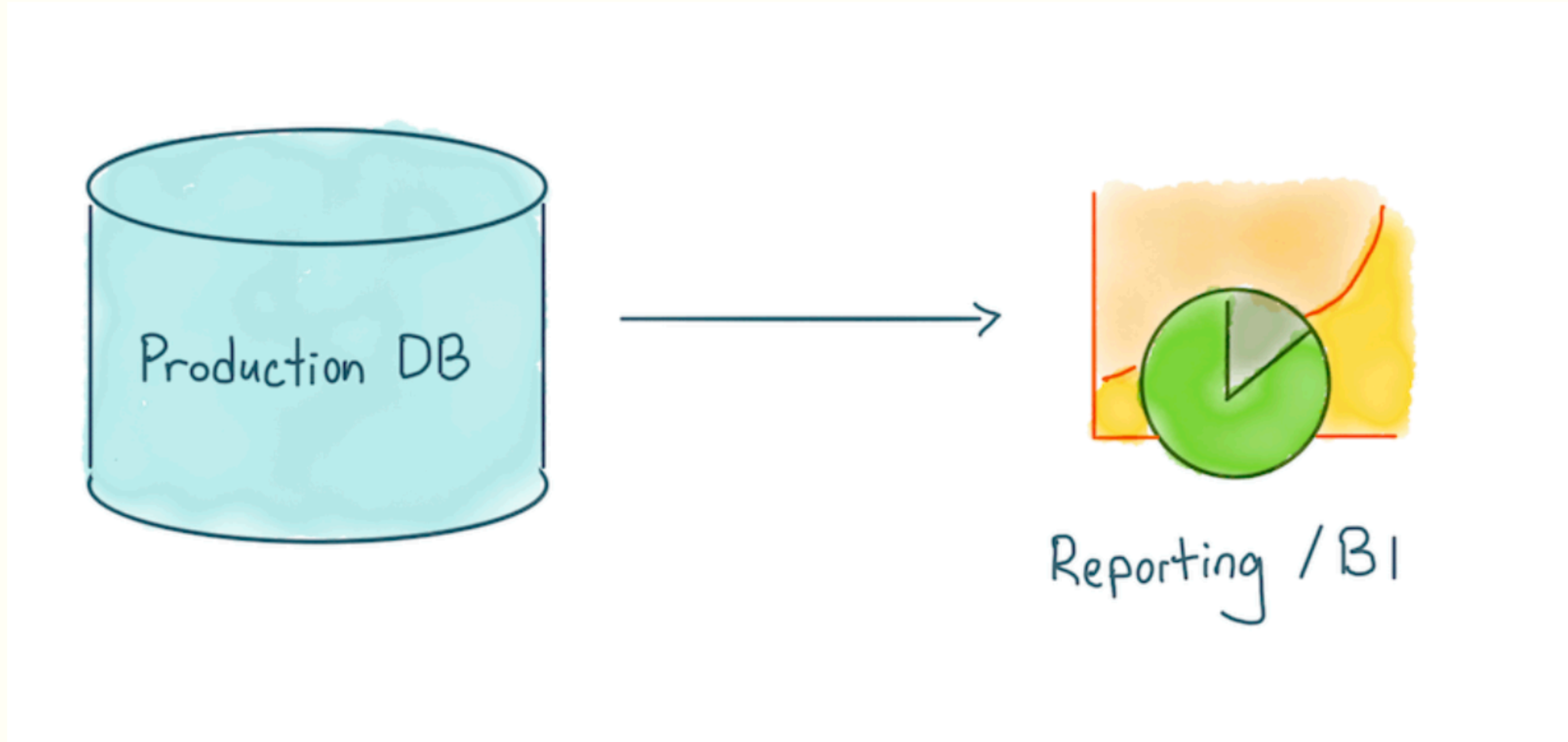
- **Dirty read**s: One client reads another client's writes before they have been committed. This could mean you see a value that would be later rolled back.

- **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.

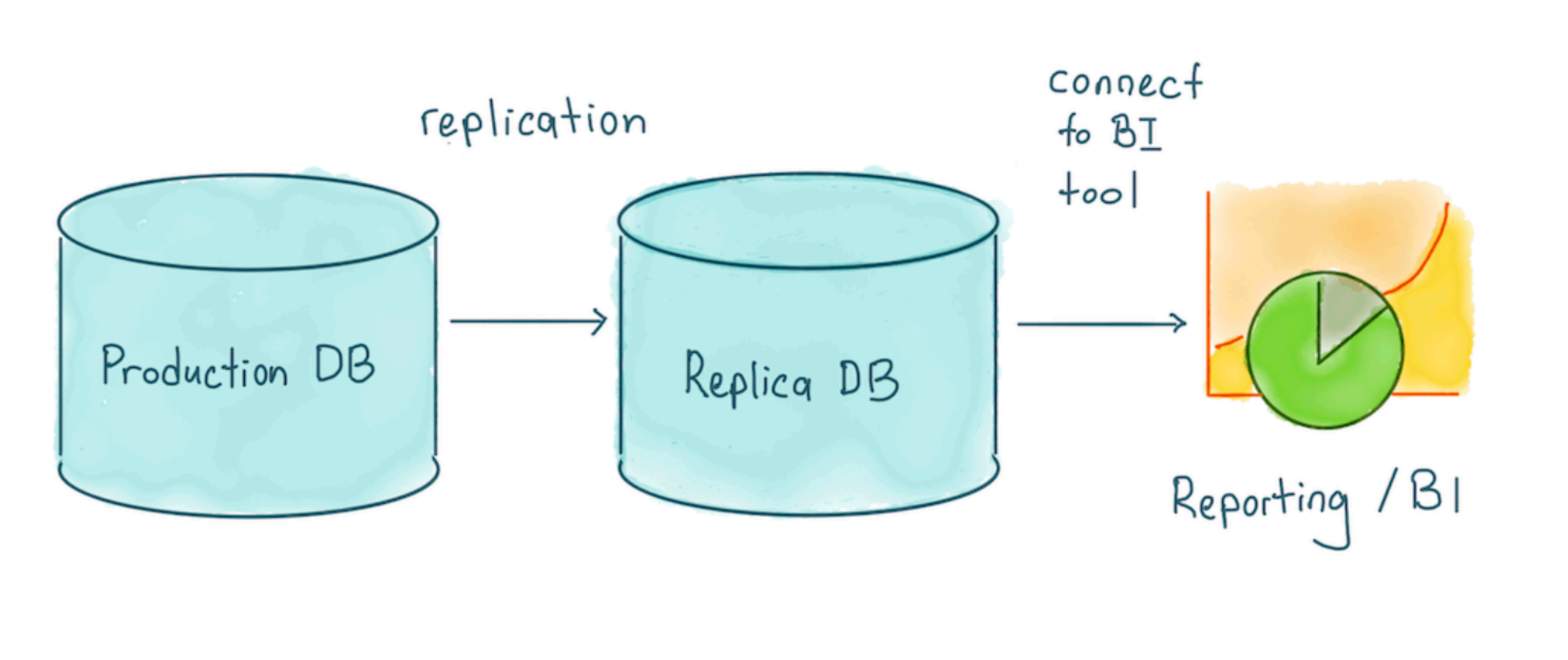
- Clearly, the notions of isolation are really the notions of concurrency: these issues will also occur when 2 programs access any data, in memory or in a database. In both cases locks and other ideas must be used to make sure that there is only one mutator at a time, and that an object is not exposed in an inconsistent state.

From transactions to analysis

Organizational Evolution

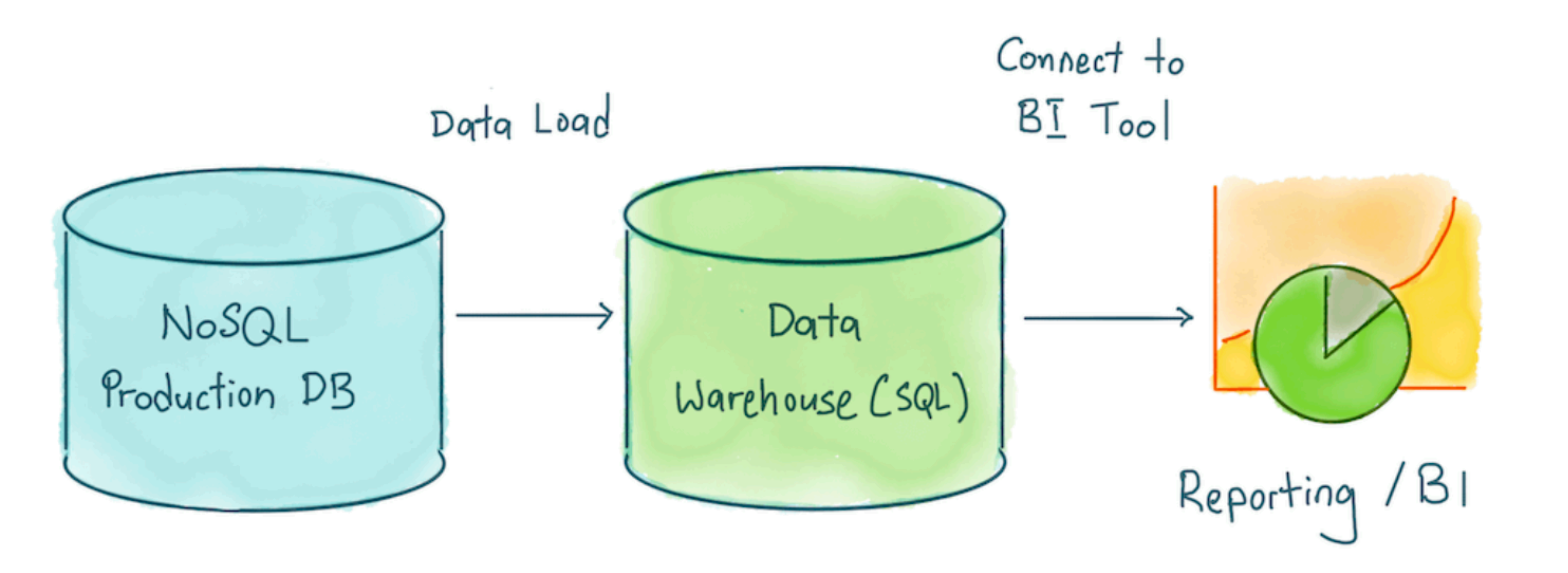


Start by using the production database as your analytics database

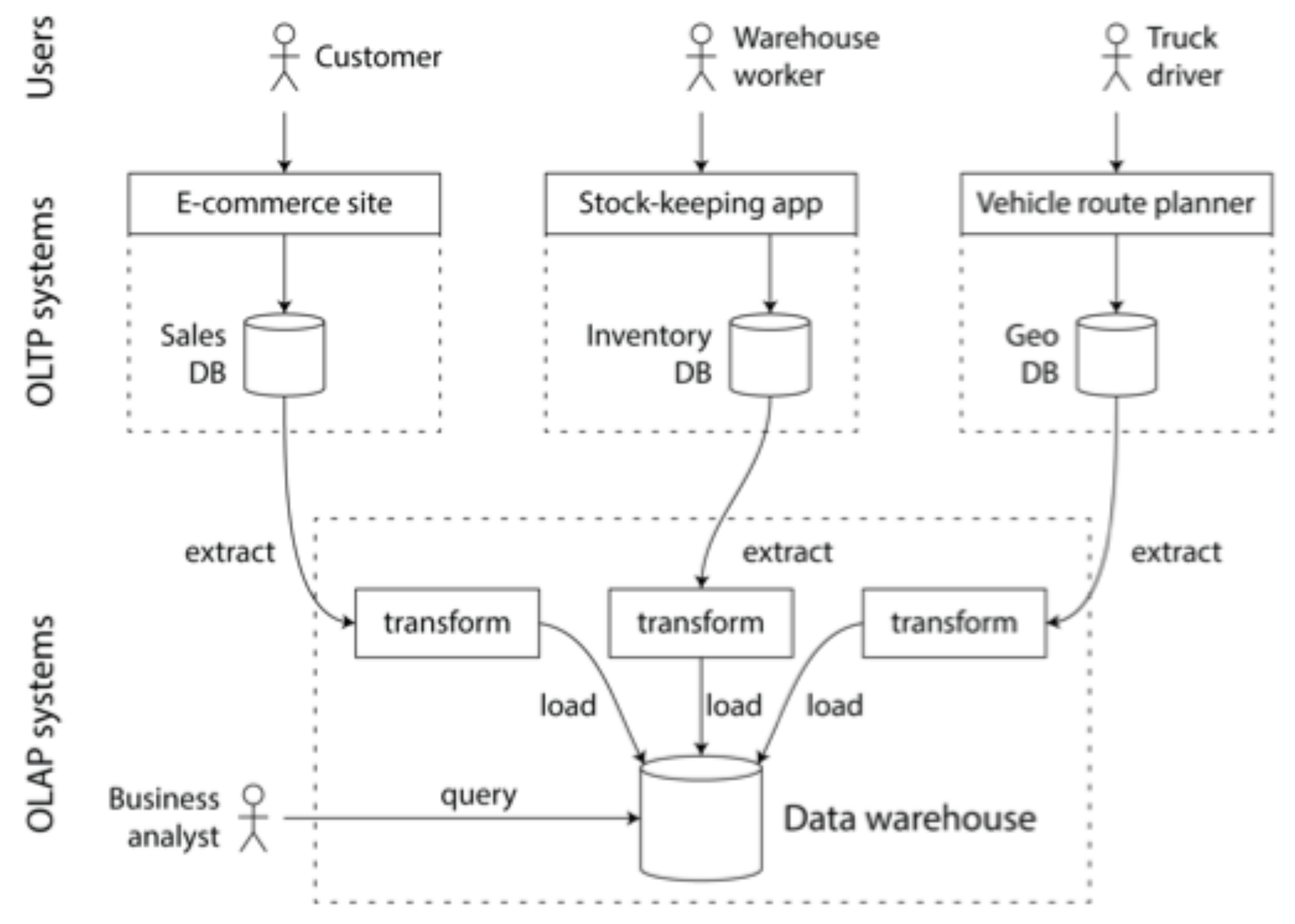


Move to using a replica of the production database as your analytics database

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

Transactional DBs	Analytics DBs
Data:	**Data:**
• Many single-row writes	• Few large batch imports
• Current, single data	• Years of data, many sources
Queries:	**Queries:**
• Generated by user activities; 10 to 1000 users	• Generated by large reports; 1 to 10 users
• < 1s response time	• Queries run for hours
• Short queries	• 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

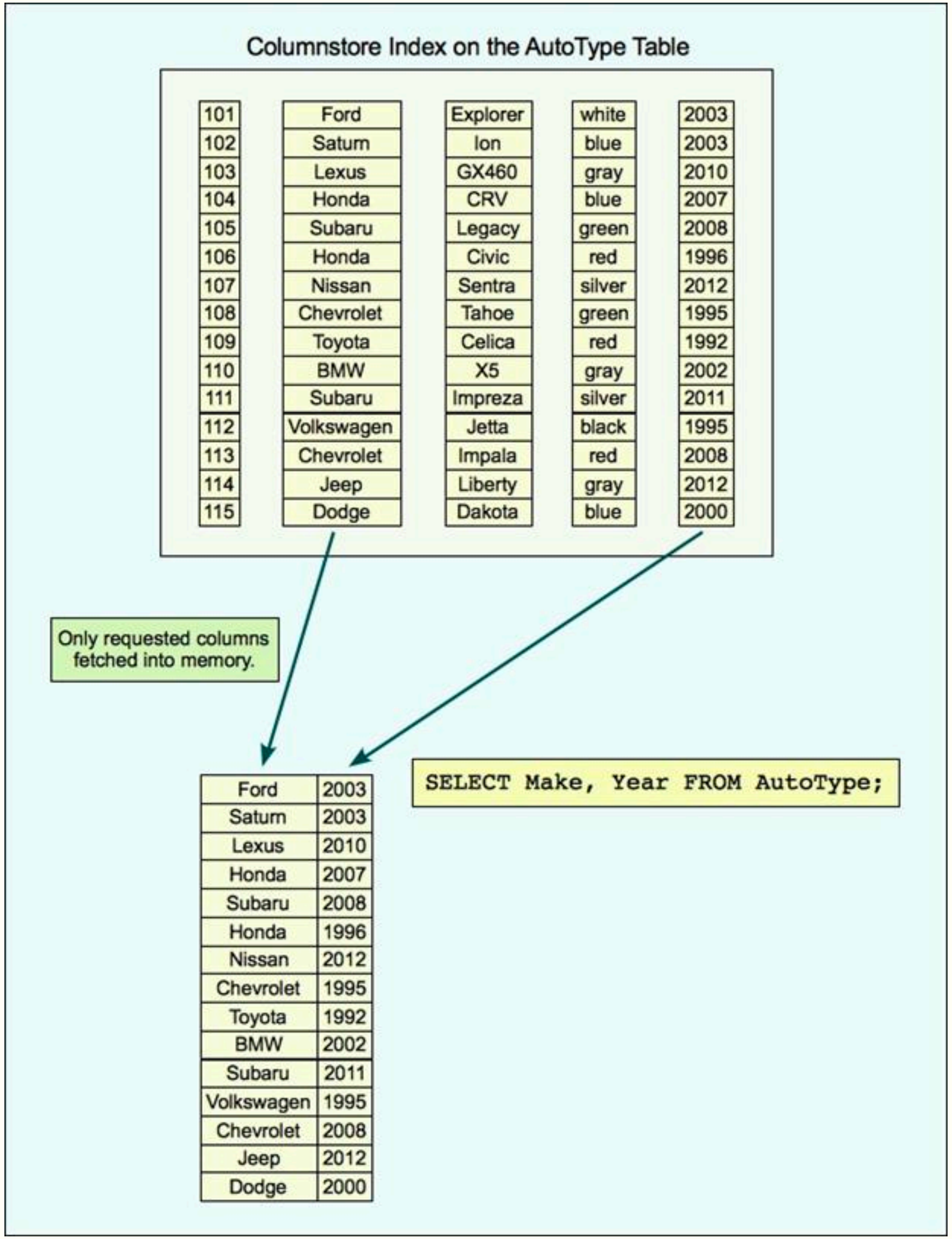
Block 1	1, 1258, US, 55.25
Block 2	2, 5698, AUS, 125.36
Block 3	3, 2265, US, 776.95
Block 4	4, 8954, CA, 32.16

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

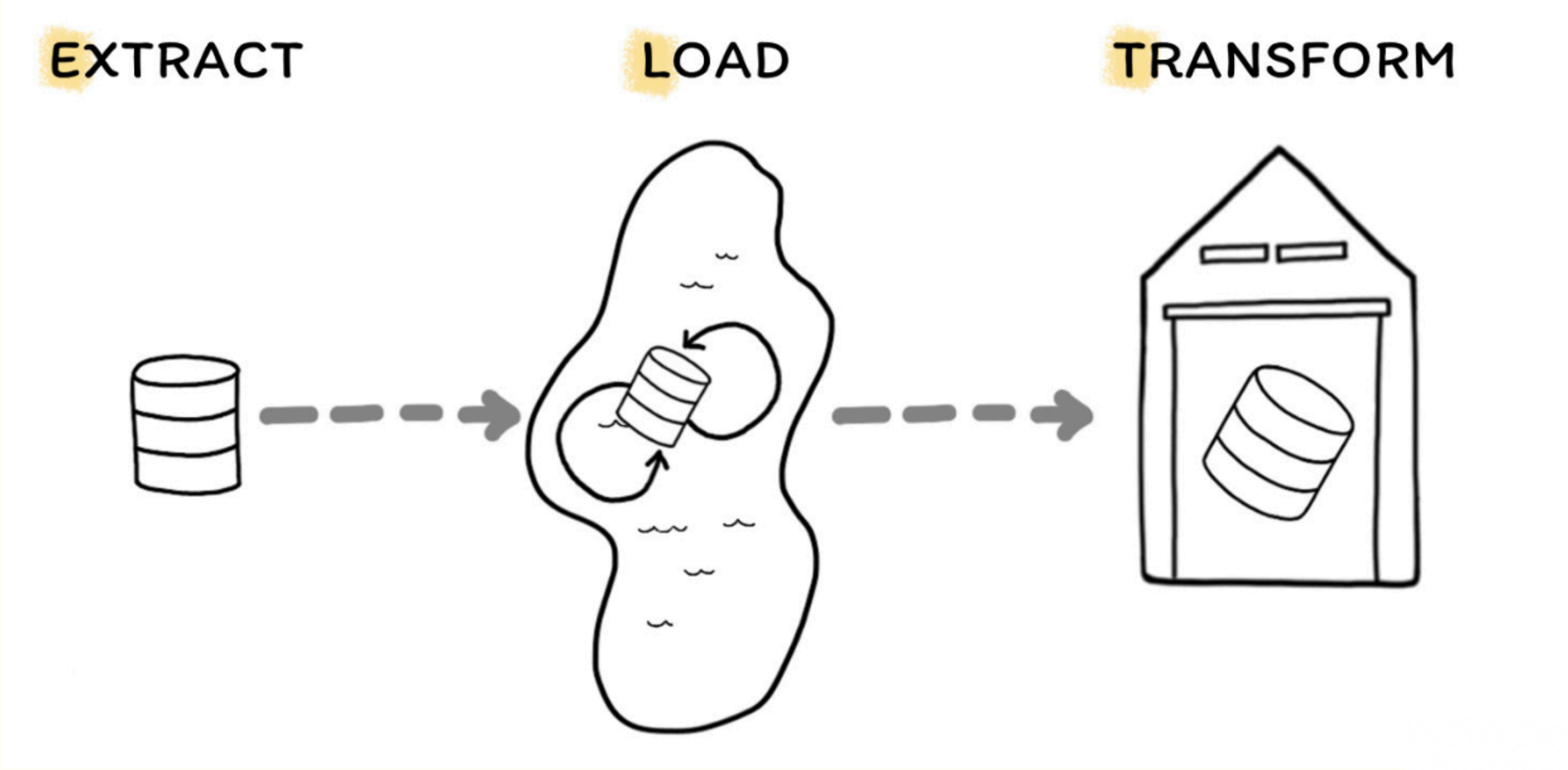
Bitmap for each possible value:

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

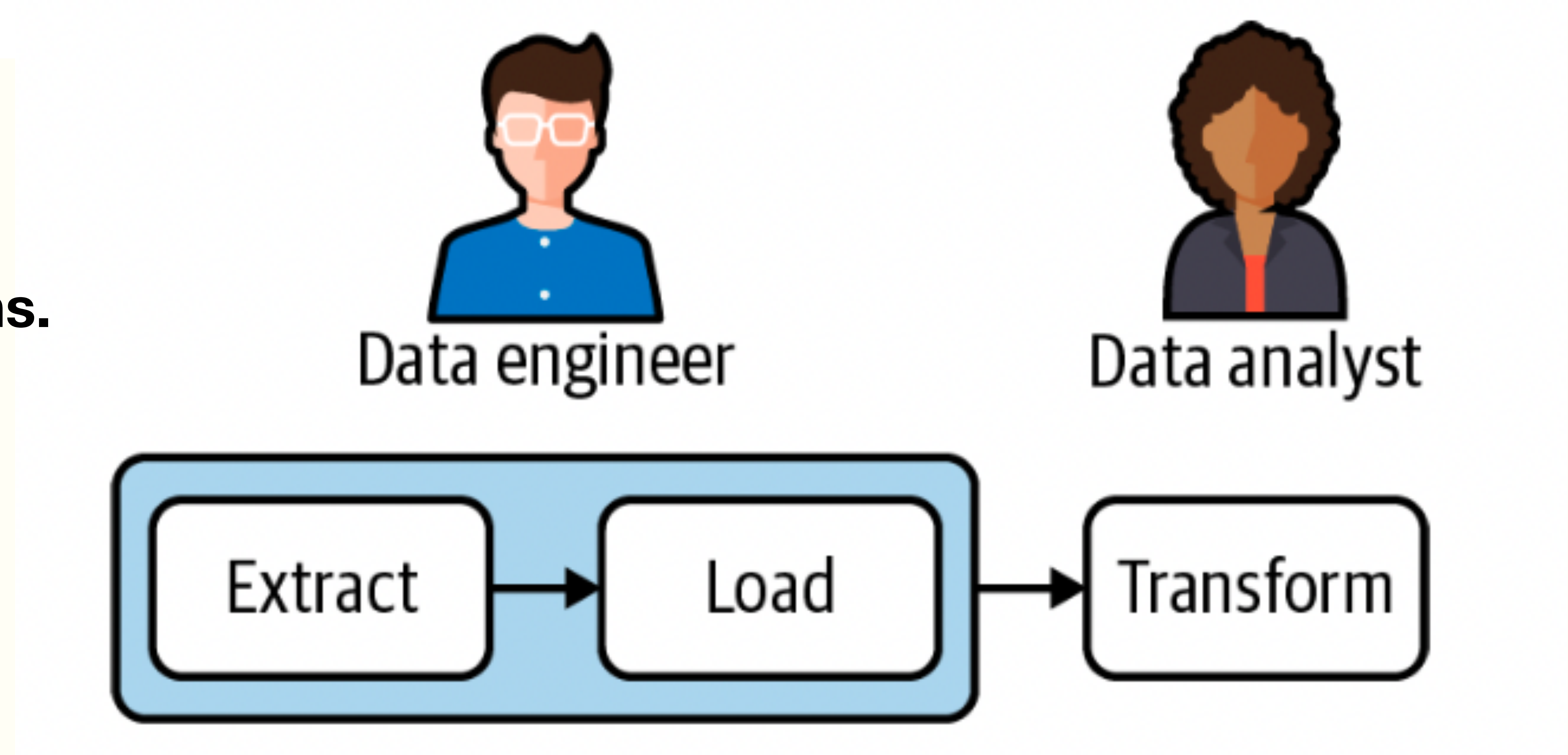
Run-length encoding:

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