OLTP Databases and SQL

Rahul Dave (@rahuldave), Univ.Ai

Univ.AI

Outline

- SQL and Normalization
- Structure of a RDBMS
- Transactions and ACID

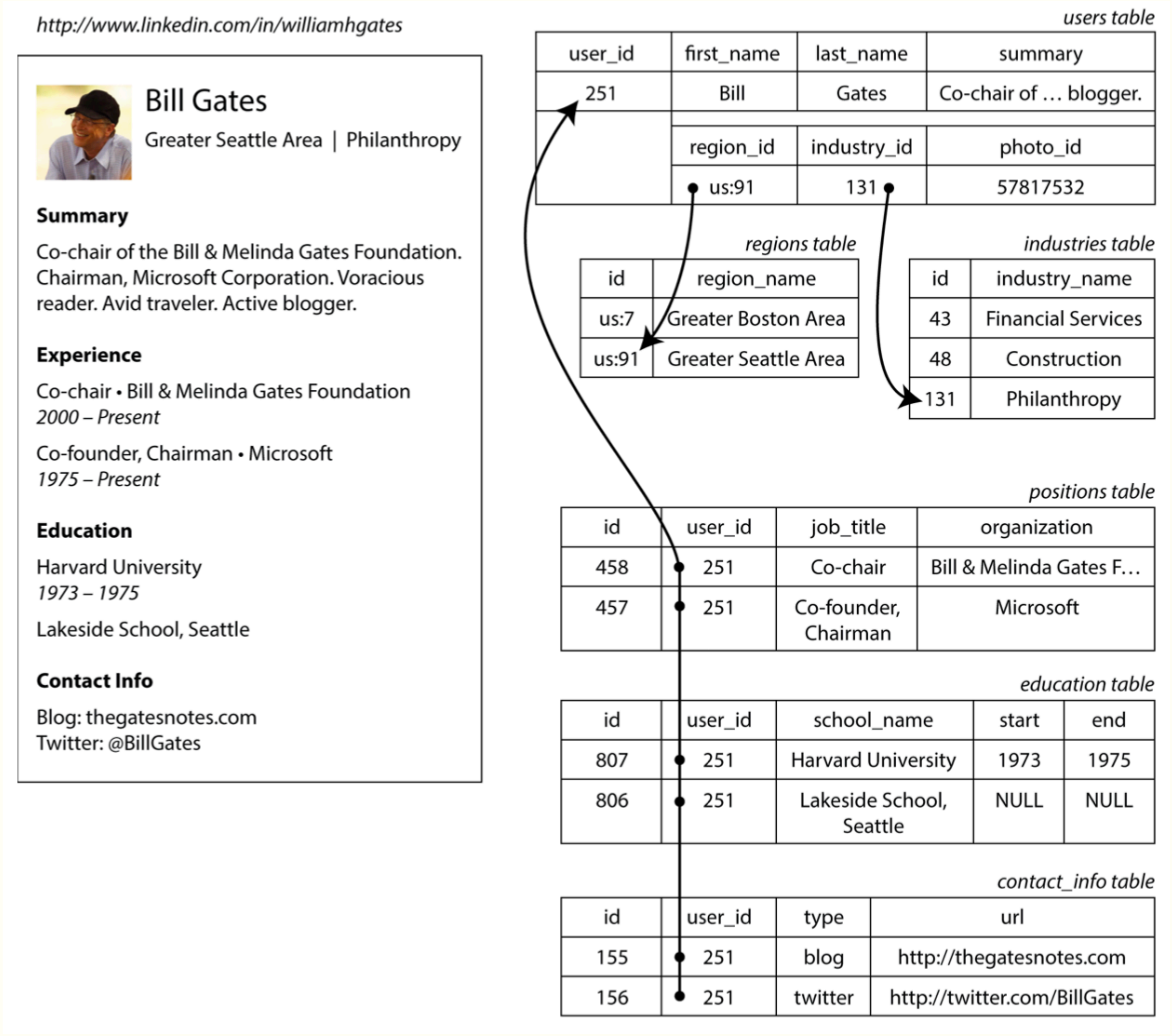
1. RDBMS and SQL

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.*



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

Lakeside School, Seattle

Contact Info

Blog: thegatesnotes.com
Twitter: @BillGates

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…
457	251	Co-founder, Chairman	Microsoft

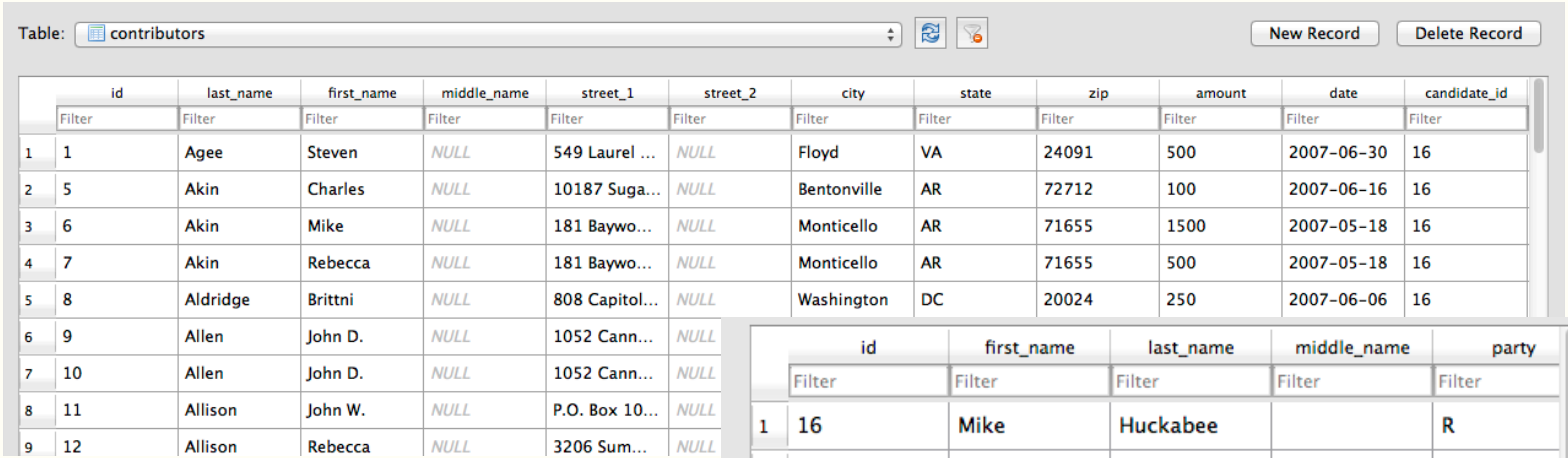
education table

id	user_id	school_name	start	end
807	251	Harvard University	1973	1975
806	251	Lakeside School, Seattle	NULL	NULL

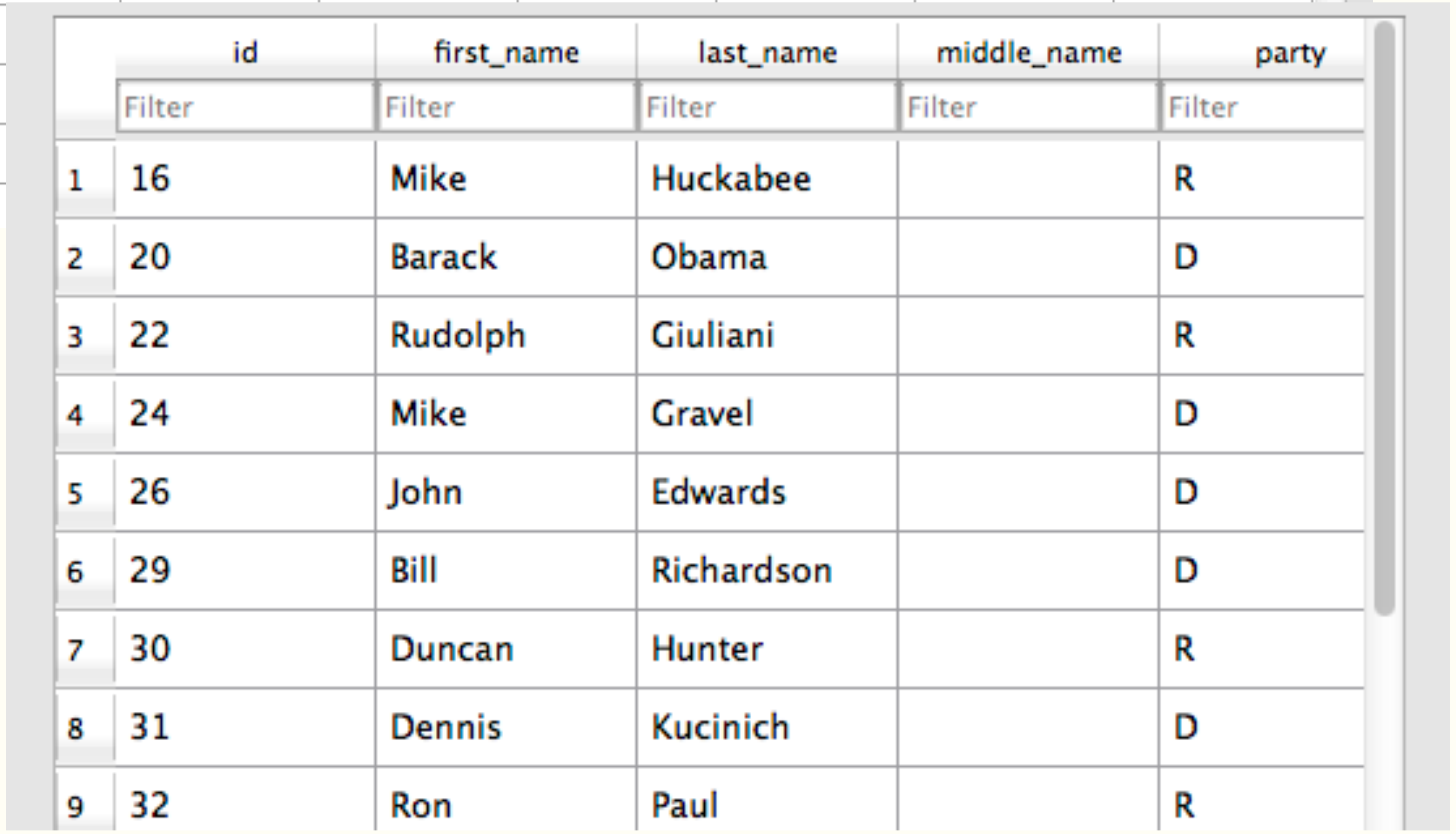
contact_info table

id	user_id	type	url
155	251	blog	http://thegatesnotes.com
156	251	twitter	http://twitter.com/BillGates

Univ.AI

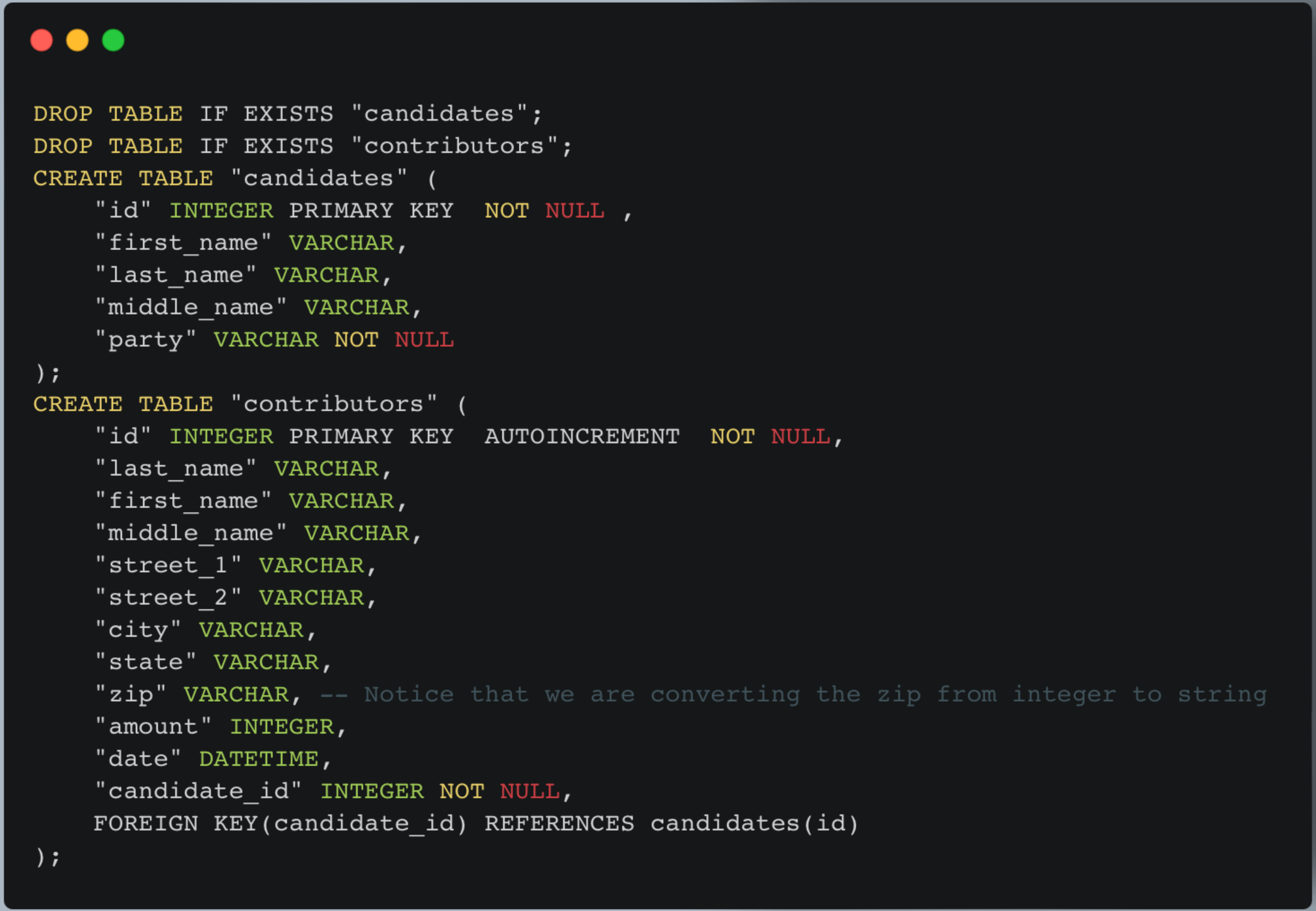


	id	last_name	first_name	middle_name	street_1	street_2	city	state	zip	amount	date	candidate_id
1	1	Agee	Steven	NULL	549 Laurel ...	NULL	Floyd	VA	24091	500	2007-06-30	16
2	5	Akin	Charles	NULL	10187 Suga...	NULL	Bentonville	AR	72712	100	2007-06-16	16
3	6	Akin	Mike	NULL	181 Baywo...	NULL	Monticello	AR	71655	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						



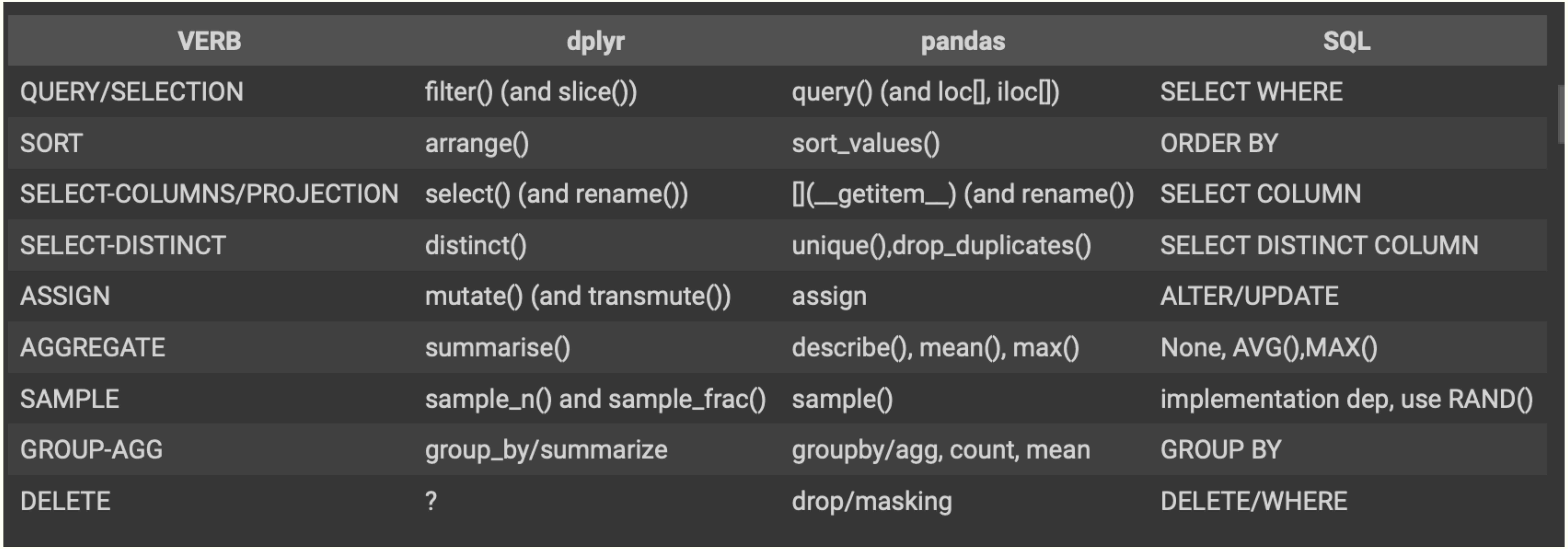
	id	first_name	last_name	middle_name	party
1	16	Mike	Huckabee		R
2	20	Barack	Obama		D
3	22	Rudolph	Giuliani		R
4	24	Mike	Gravel		D
5	26	John	Edwards		D
6	29	Bill	Richardson		D
7	30	Duncan	Hunter		R
8	31	Dennis	Kucinich		D
9	32	Ron	Paul		R

Create Tables



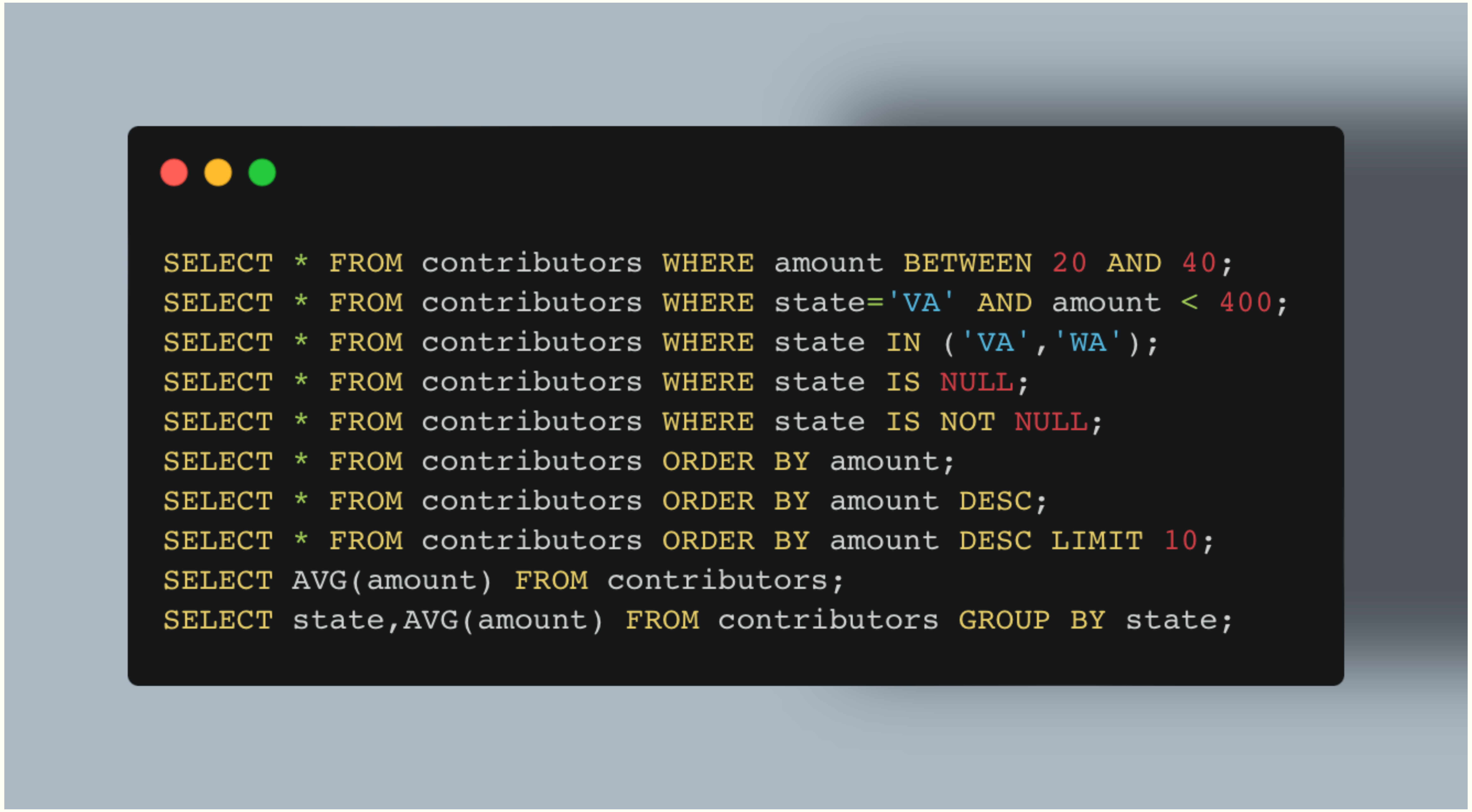
```
DROP TABLE IF EXISTS "candidates";
DROP TABLE IF EXISTS "contributors";
CREATE TABLE "candidates" (
    "id" INTEGER PRIMARY KEY  NOT NULL ,
    "first_name" VARCHAR,
    "last_name" VARCHAR,
    "middle_name" VARCHAR,
    "party" VARCHAR NOT NULL
);
CREATE TABLE "contributors" (
    "id" INTEGER 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)
);
```

Verbs



VERB	dplyr	pandas	SQL
QUERY/SELECTION	filter() (and slice())	query() (and loc[], iloc[])	SELECT WHERE
SORT	arrange()	sort_values()	ORDER BY
SELECT-COLUMNS/PROJECTION	select() (and rename())	[](__getitem__) (and rename())	SELECT COLUMN
SELECT-DISTINCT	distinct()	unique(),drop_duplicates()	SELECT DISTINCT COLUMN
ASSIGN	mutate() (and transmute())	assign	ALTER/UPDATE
AGGREGATE	summarise()	describe(), mean(), max()	None, AVG(),MAX()
SAMPLE	sample_n() and sample_frac()	sample()	implementation dep, use RAND()
GROUP-AGG	group_by/summarize	groupby/agg, count, mean	GROUP BY
DELETE	?	drop/masking	DELETE/WHERE

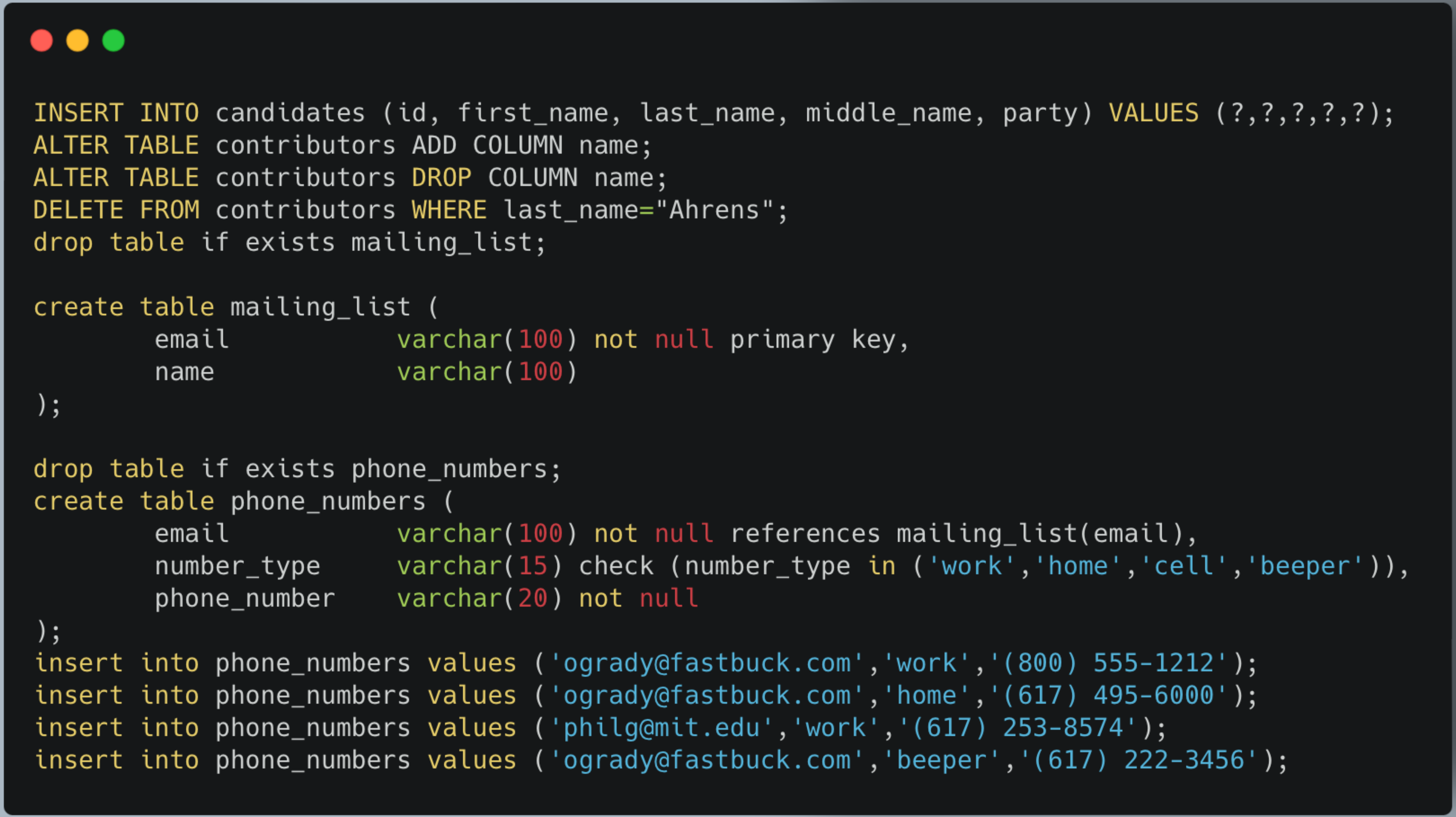
selects in tables



```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;
```

Univ.AI

inserts and alters

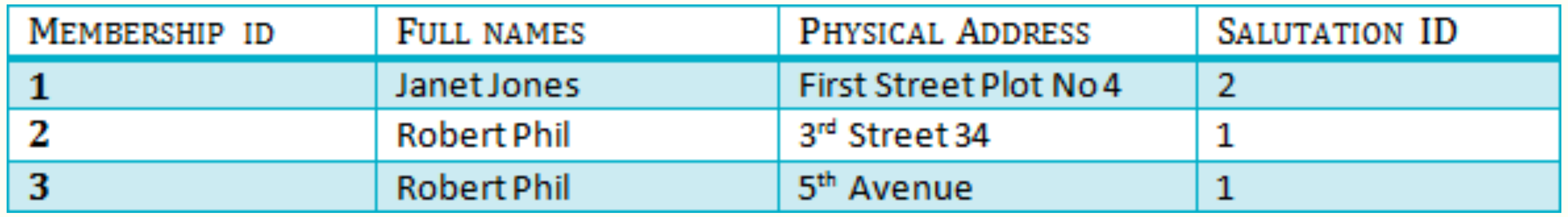


```
INSERT INTO candidates (id, first_name, last_name, middle_name, party) VALUES (?,?,?,?,?);
ALTER TABLE contributors ADD COLUMN name;
ALTER TABLE contributors DROP COLUMN name;
DELETE FROM contributors WHERE last_name="Ahrens";
drop table if exists mailing_list;

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');
```

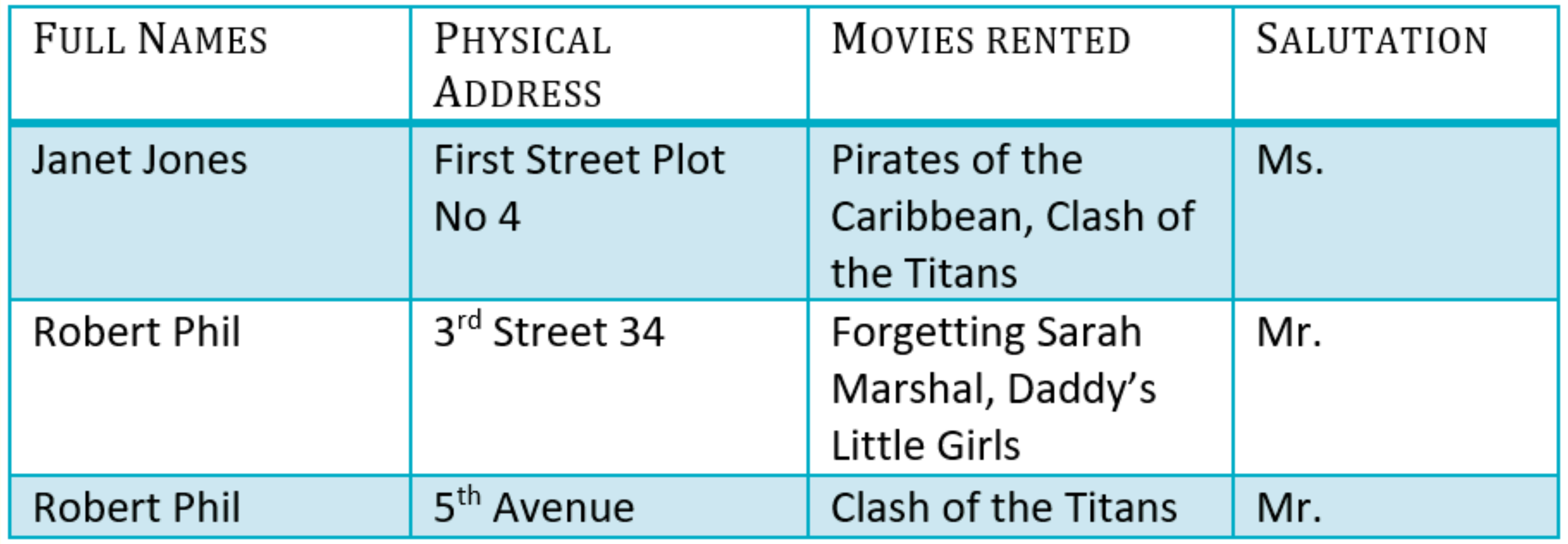
Univ.AI

Normalization

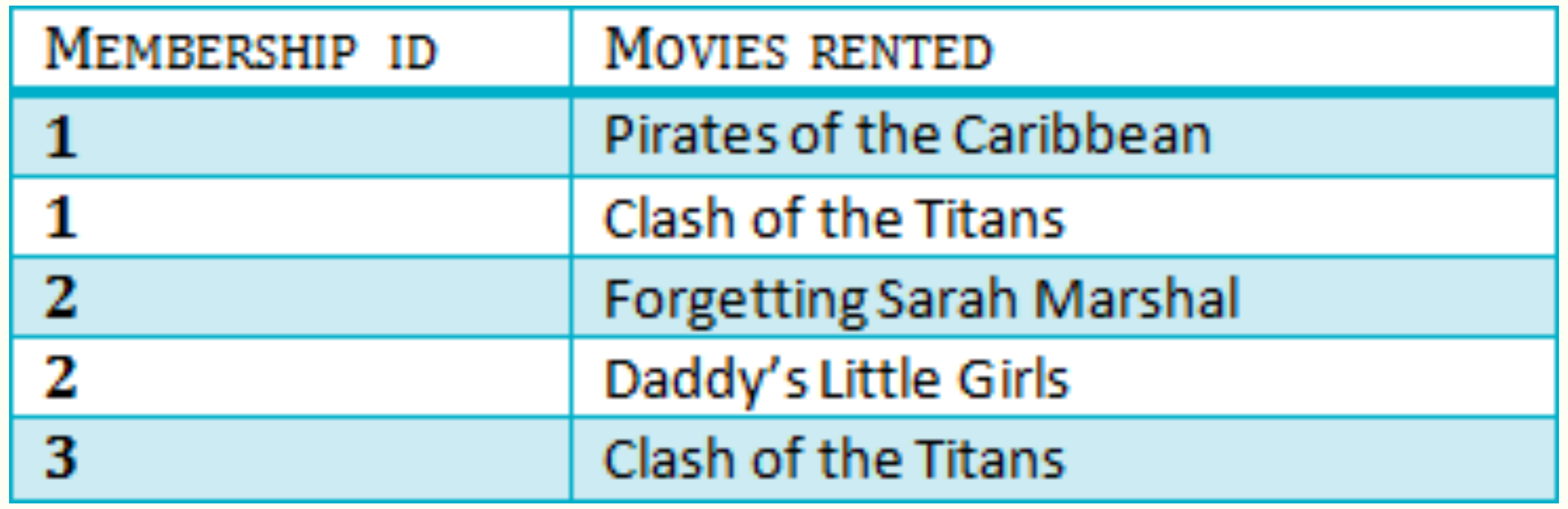
Make your tables DRY. Details on Normal Forms: https://www.guru99.com/database-normalization.html



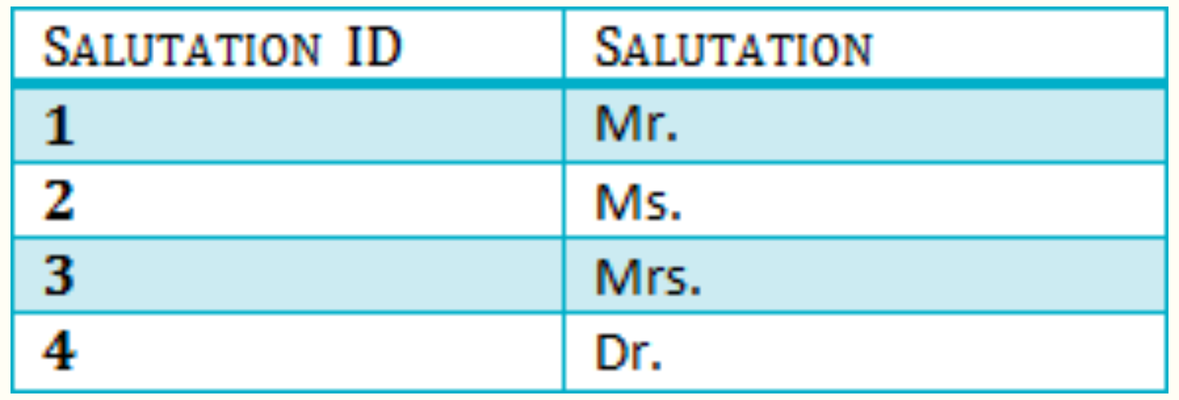
Membership ID	Full names	Physical Address	Salutation ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1



Full Names	Physical Address	Movies rented	Salutation
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.



Membership ID	Movies rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans



Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

1NF: table cells/cols unique type, each row unique **2NF**: single column primary key **3NF**: no transitive functional dependencies

Univ.AI

Normalization

Full Names	Physical Address	Movies rented	Salutation
Janet Jones	First Street Plot No 4	Pirates of the Caribbean, Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal, Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

Full Names	Physical Address	Movies rented	Salutation
Janet Jones	First Street Plot No 4	Pirates of the Caribbean	Ms.
Janet Jones	First Street Plot No 4	Clash of the Titans	Ms.
Robert Phil	3rd Street 34	Forgetting Sarah Marshal	Mr.
Robert Phil	3rd Street 34	Daddy's Little Girls	Mr.
Robert Phil	5th Avenue	Clash of the Titans	Mr.

1NF

Membership ID	Full Names	Physical Address	Salutation
1	Janet Jones	First Street Plot No 4	Ms.
2	Robert Phil	3rd Street 34	Mr.
3	Robert Phil	5th Avenue	Mr.

2NF

Membership ID	Movies rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Membership ID	Movies rented
1	Pirates of the Caribbean
1	Clash of the Titans
2	Forgetting Sarah Marshal
2	Daddy's Little Girls
3	Clash of the Titans

Salutation ID	Salutation
1	Mr.
2	Ms.
3	Mrs.
4	Dr.

3NF

Membership ID	Full Names	Physical Address	Salutation ID
1	Janet Jones	First Street Plot No 4	2
2	Robert Phil	3rd Street 34	1
3	Robert Phil	5th Avenue	1

1NF: table cells/cols unique type, each row unique **2NF**: single column primary key **3NF**: no transitive functional dependencies

Univ.AI

Why Normalize?

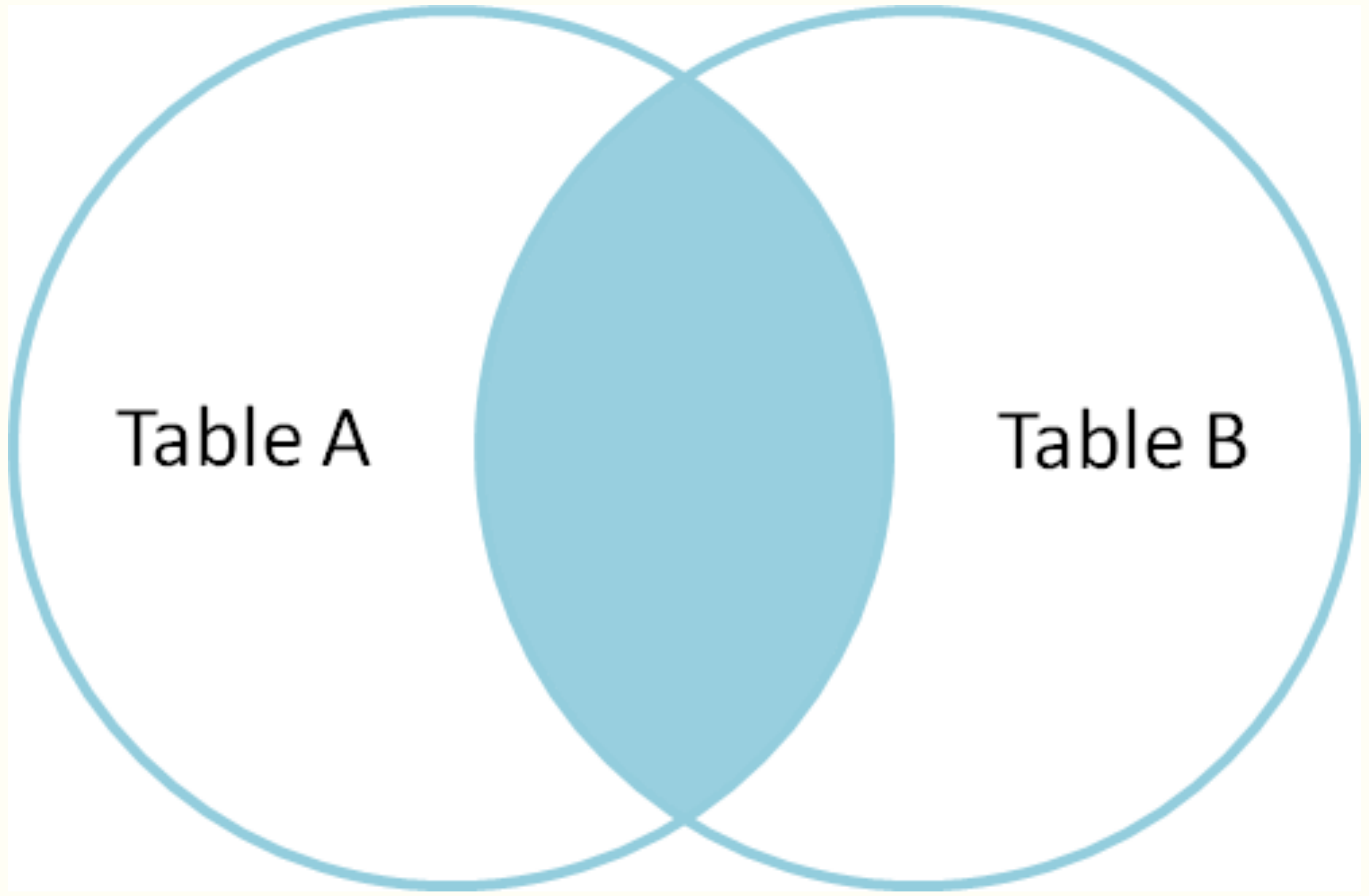
- free collection of relations from undesirable insertion, update and deletion dependencies
- mitigate restructuring throughout the project as new data is introduced
- make the relational model more informative to users
- make collection of relations neutral to the query statistics

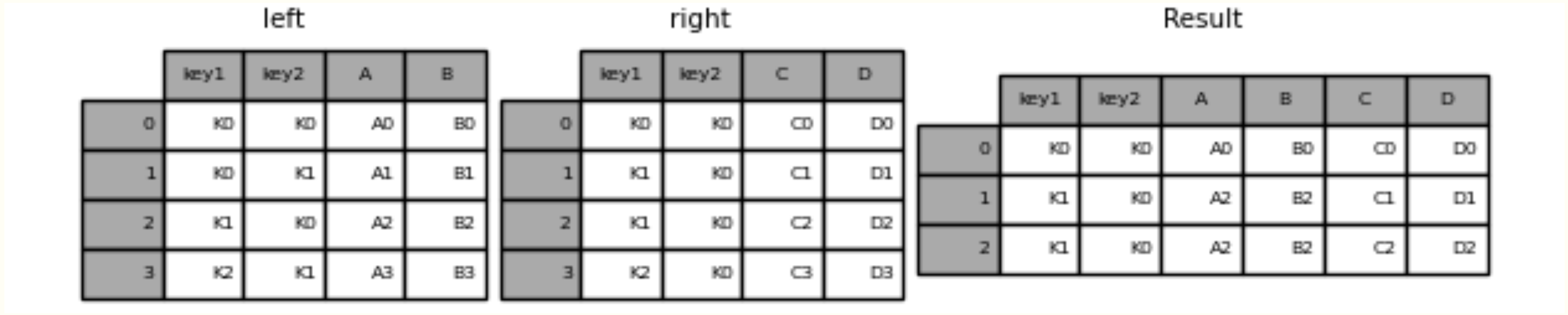
partial dependency occurs when a subset of fields in a composite key can be used to determine a non-key column of the table

1. Denormalized: No normalization. nested and redundant data is allowed
2. First Normal Form (1NF): Each column is unique and has a single value. The table has a unique primary key.
3. Second Normal Form (2NF): The requirements of 1NF, plus partial dependencies are removed.
4. Third Normal Form (3NF): The requirements of 2NF, plus each table contains only relevant fields related to its primary key and has no transitive dependencies.

A transitive dependency occurs when a non-key field depends on another non-key field

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

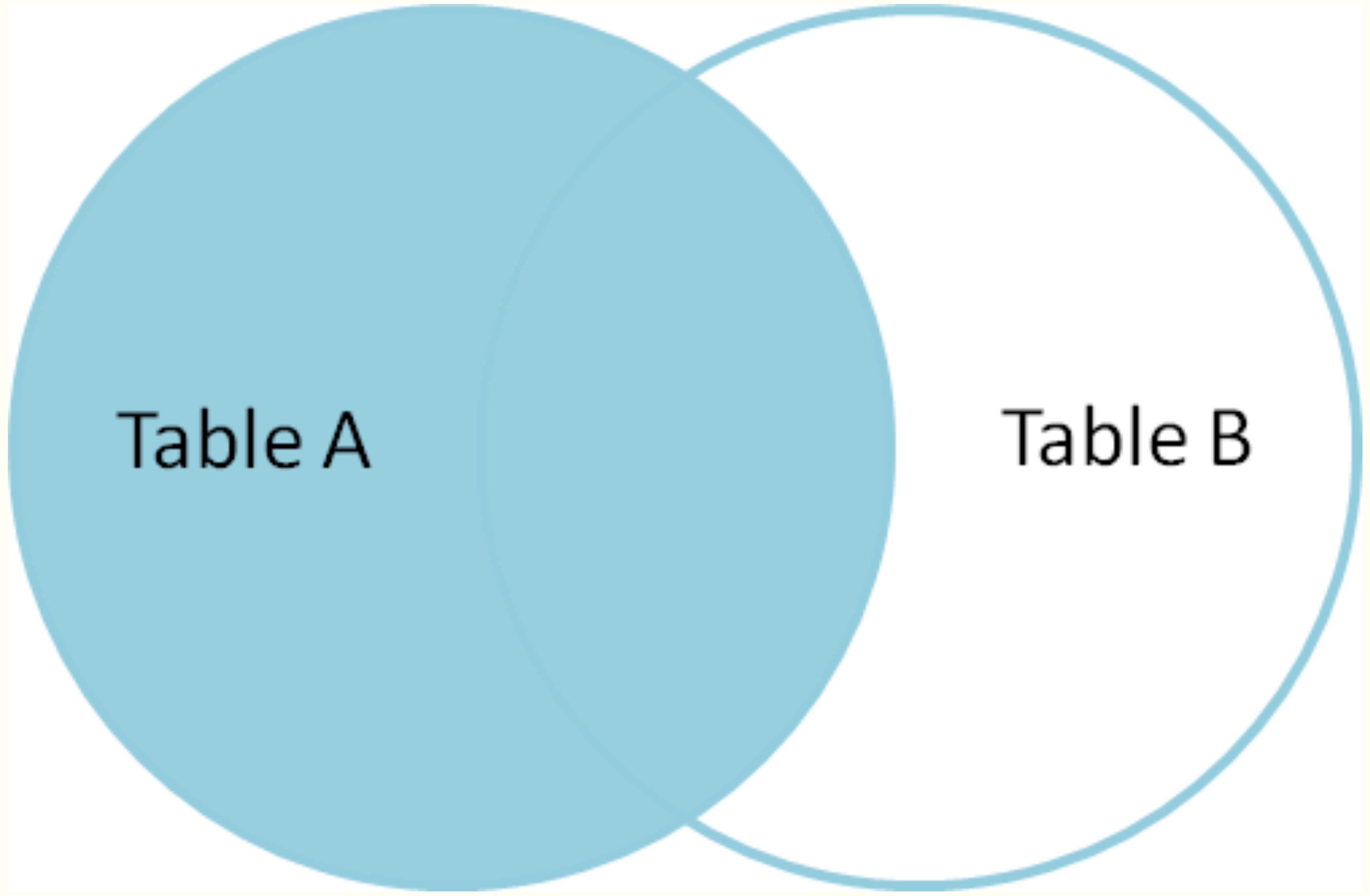
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

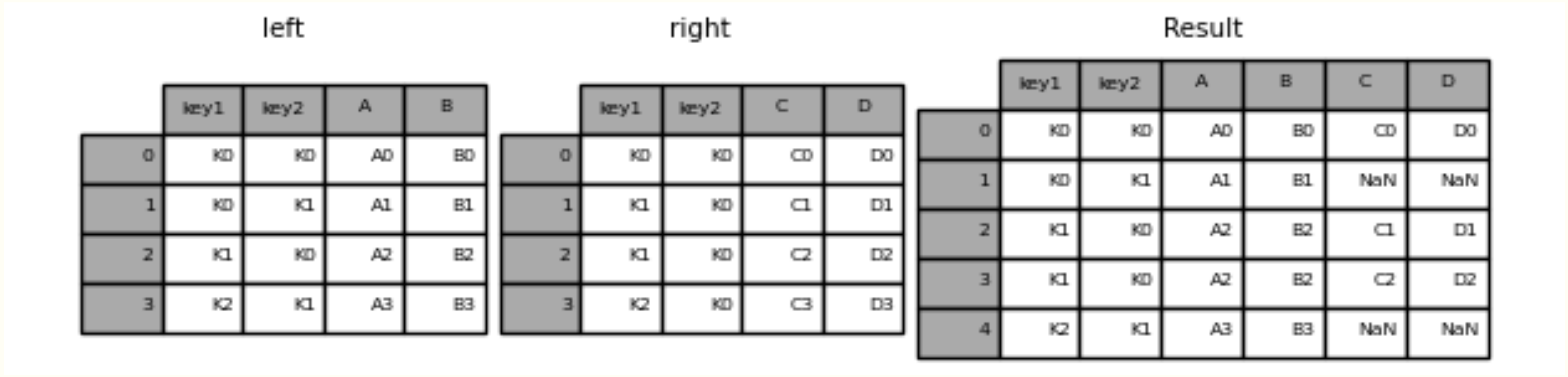
Result

	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

Univ.AI

Outer Join (left)





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

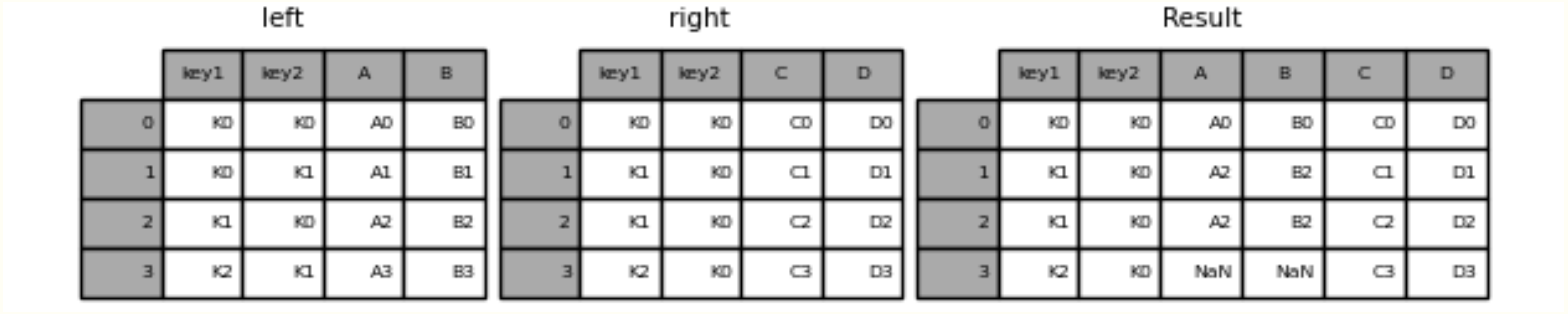
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

Result

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN

Univ.AI

Right Outer Join



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

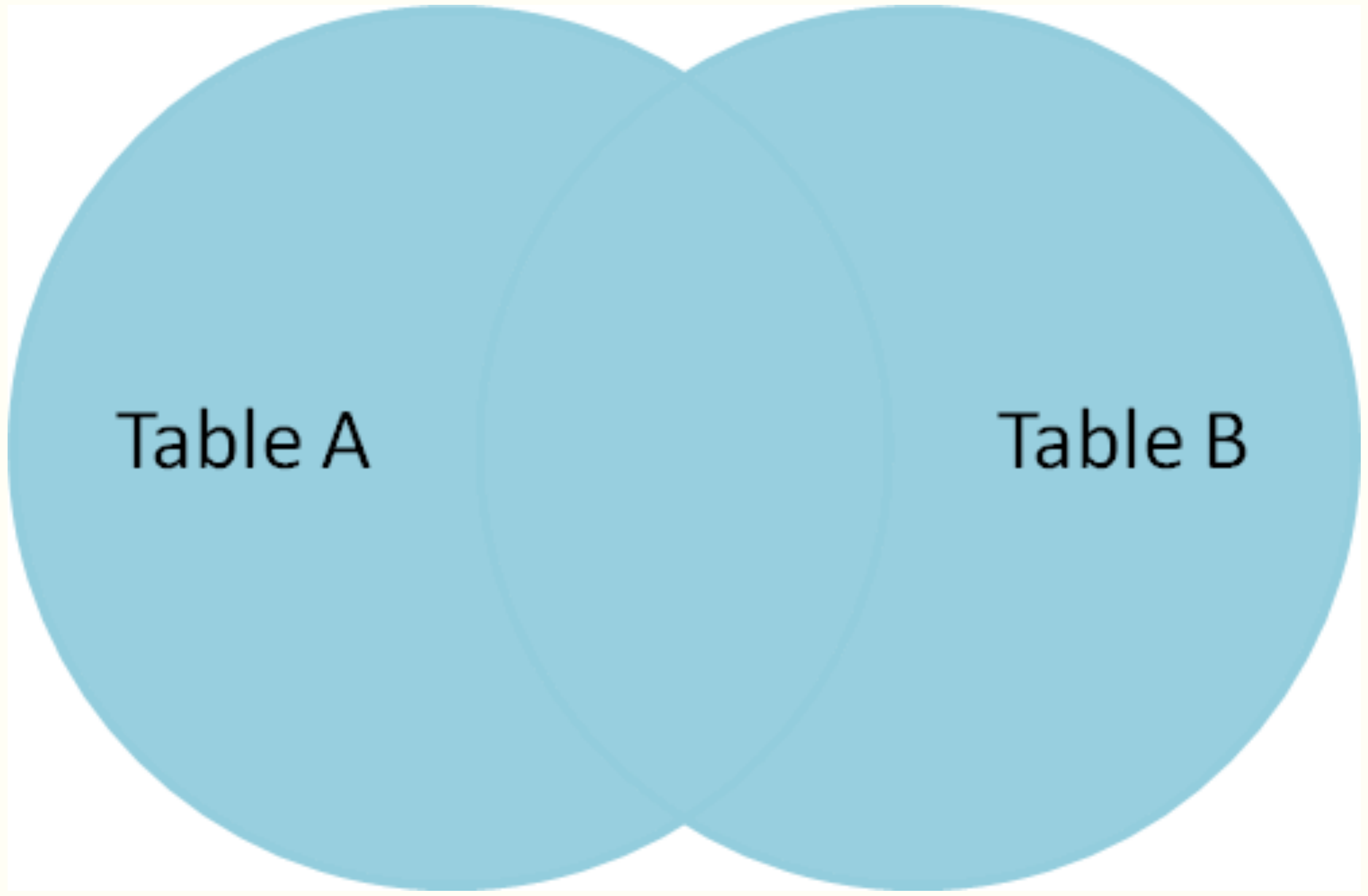
right

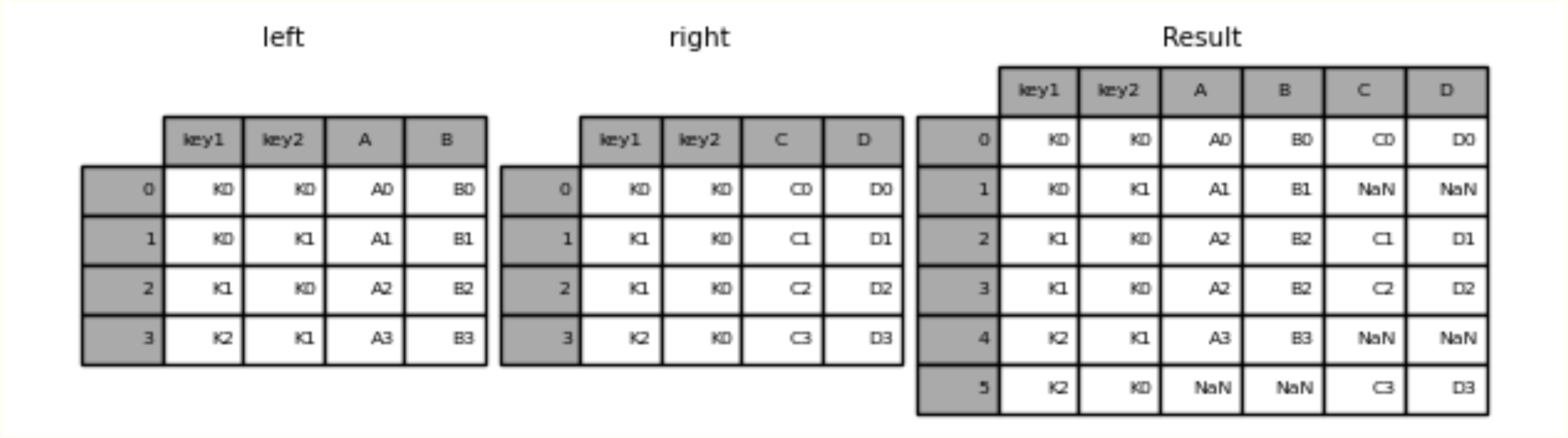
	key1	key2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K1	K0	C2	D2
3	K2	K0	C3	D3

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
3	K2	K0	NaN	NaN	C3	D3

Full Outer Join





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	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C3	D3

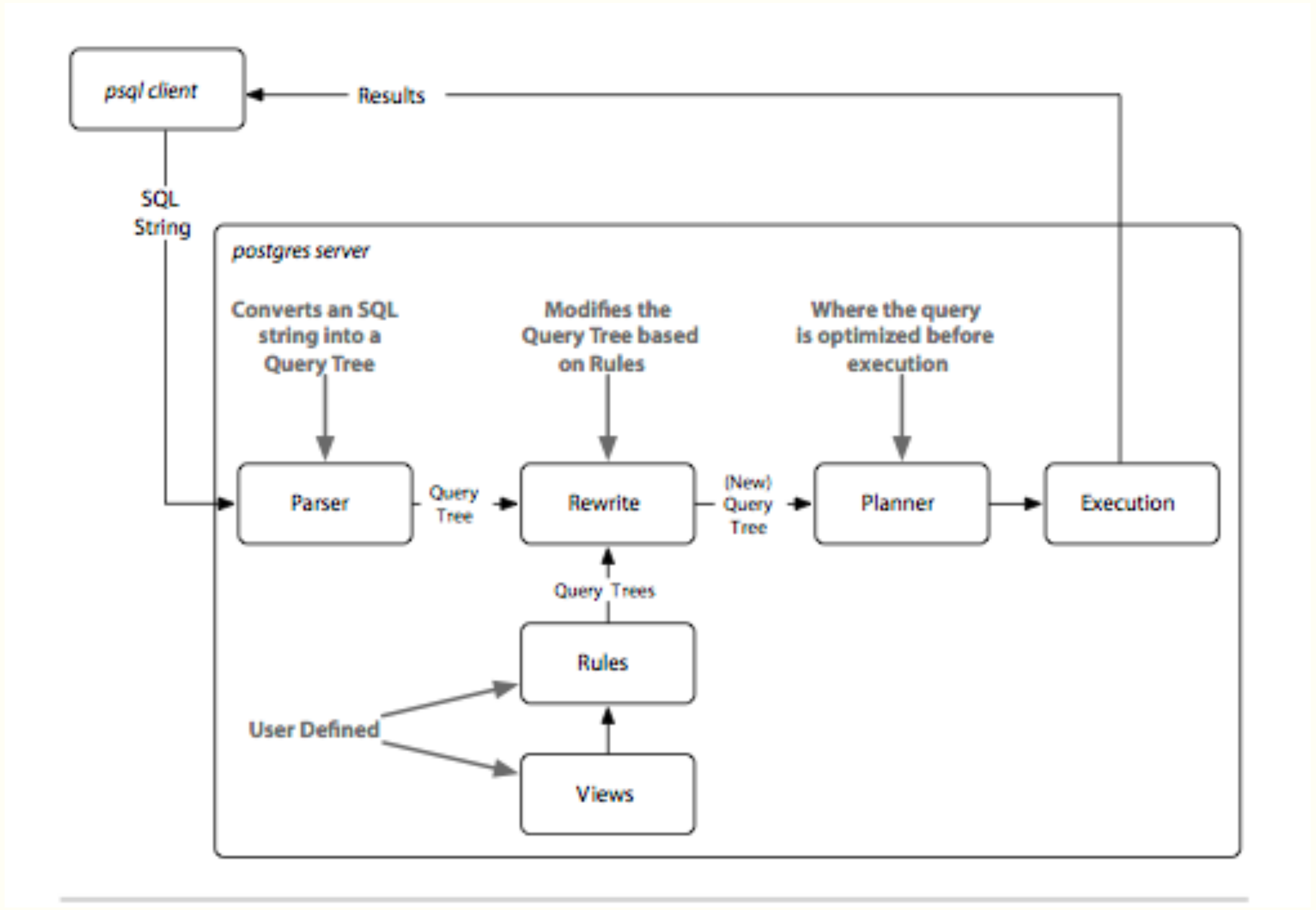
Univ.AI

2. The components of a RDBMS

Univ.AI

How does a SQL query work?

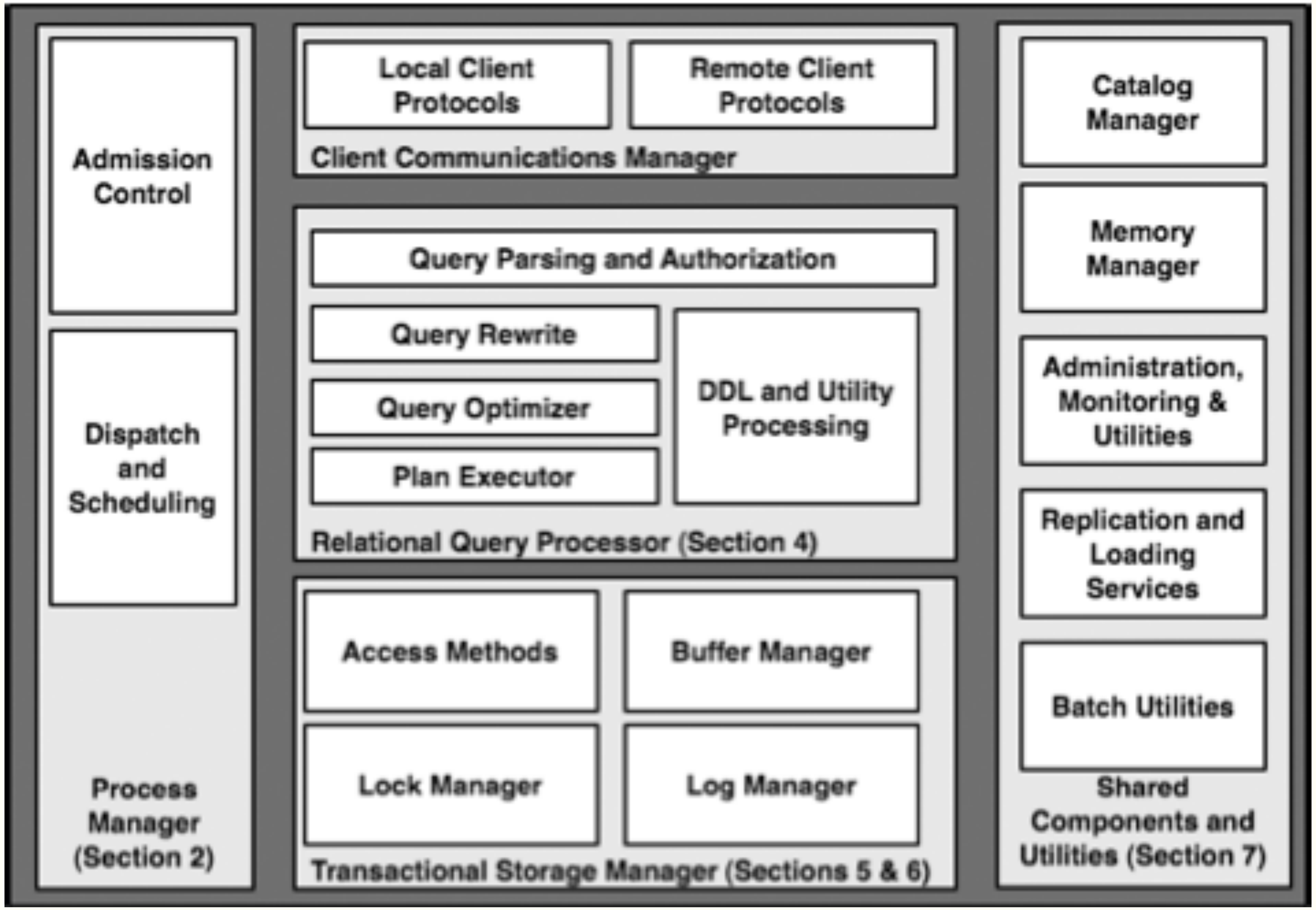
SQL is a declarative model: a query optimizer decides how to execute the query (if a field range covers 80% of values, should we use the index or the table?). Also parallelize-able.



Univ.AI

How does a RDBMS work?

- client connection manager: what to do with incomings
- transactional storage: storage data structures and the log
- process model: coroutines, threads, processes
- query model and language: query optimization


Admission Control
Dispatch and Scheduling
Process Manager (Section 2)
Local Client Protocols
Remote Client Protocols
Client Communications Manager
Query Parsing and Authorization
Query Rewrite
Query Optimizer
Plan Executor
DDL and Utility Processing
Relational Query Processor (Section 4)
Access Methods
Buffer Manager
Lock Manager
Log Manager
Transactional Storage Manager (Sections 5 & 6)
Catalog Manager
Memory Manager
Administration, Monitoring & Utilities
Replication and Loading Services
Batch Utilities
Shared Components and Utilities (Section 7)

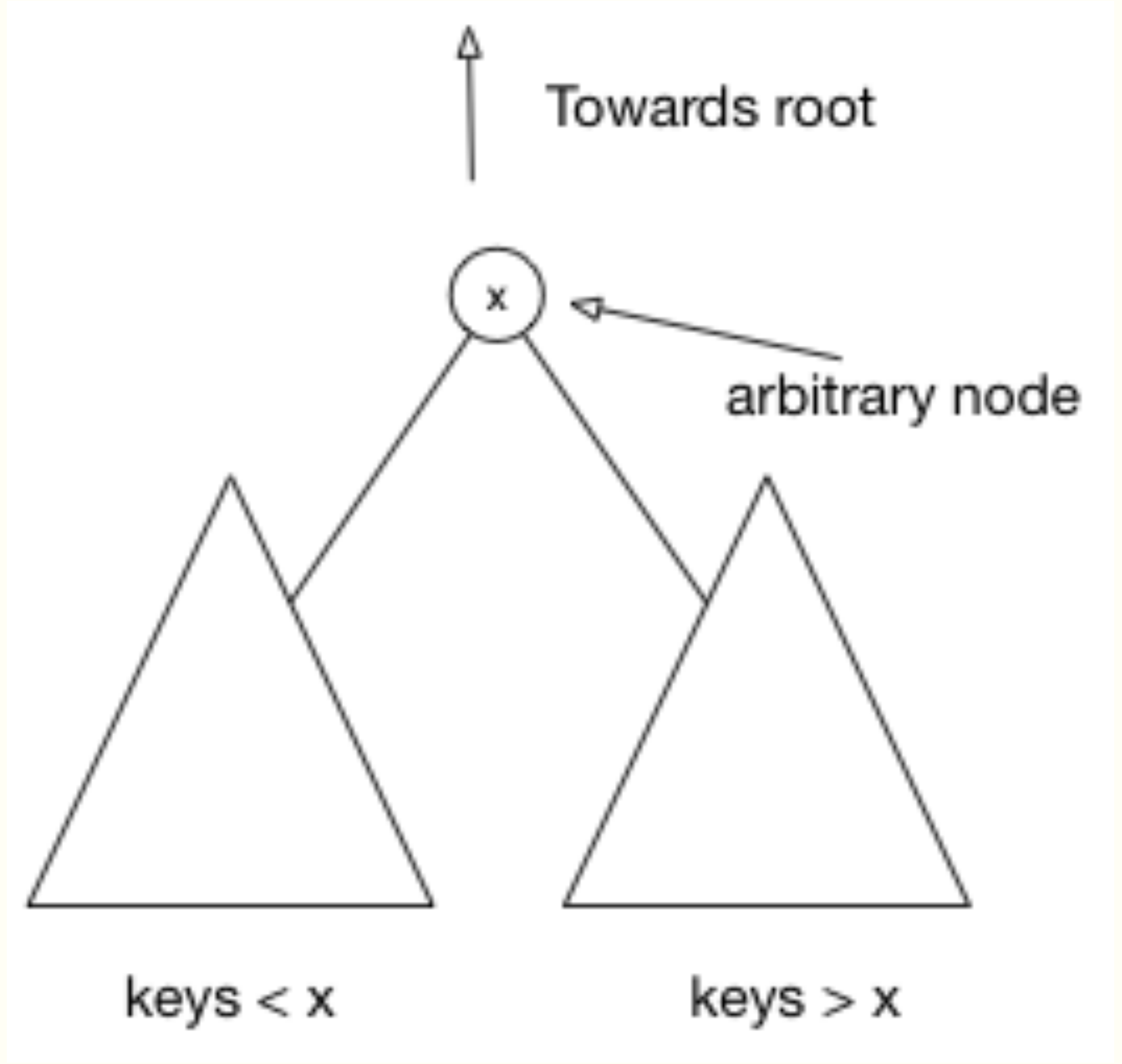
Univ.AI

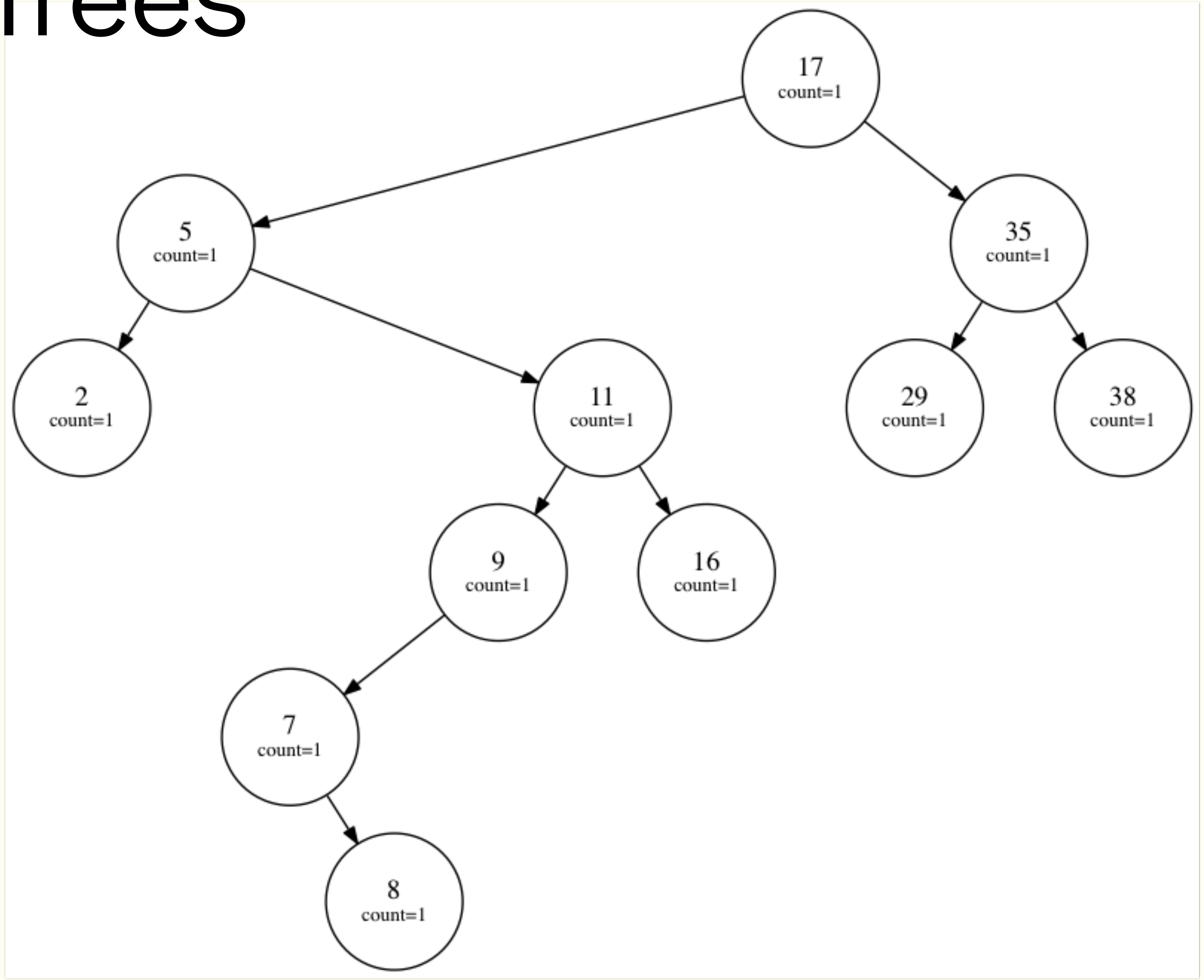
Storage Components of a RDBMS

- 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

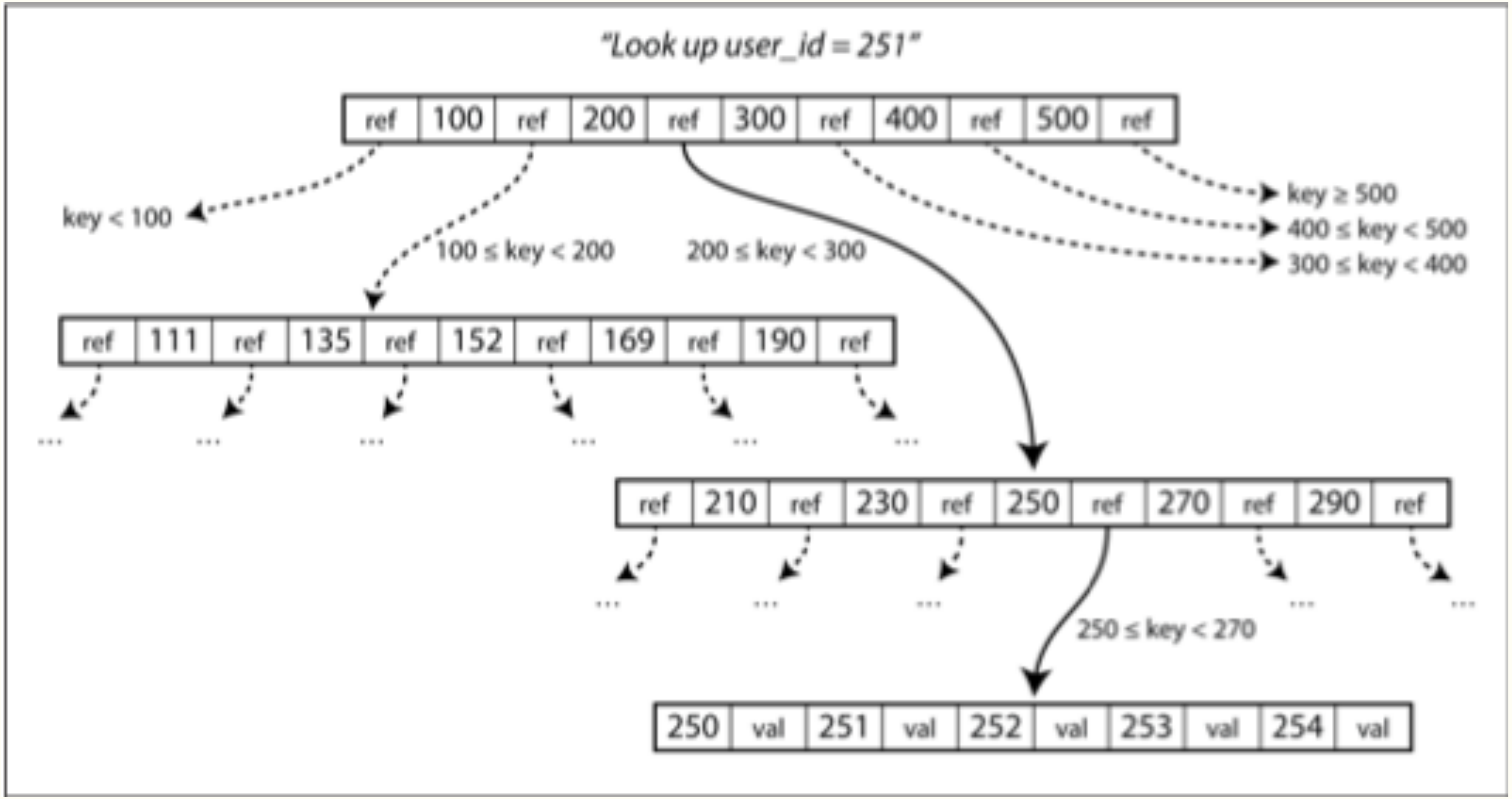
Binary Search Trees

- 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:





Univ.AI

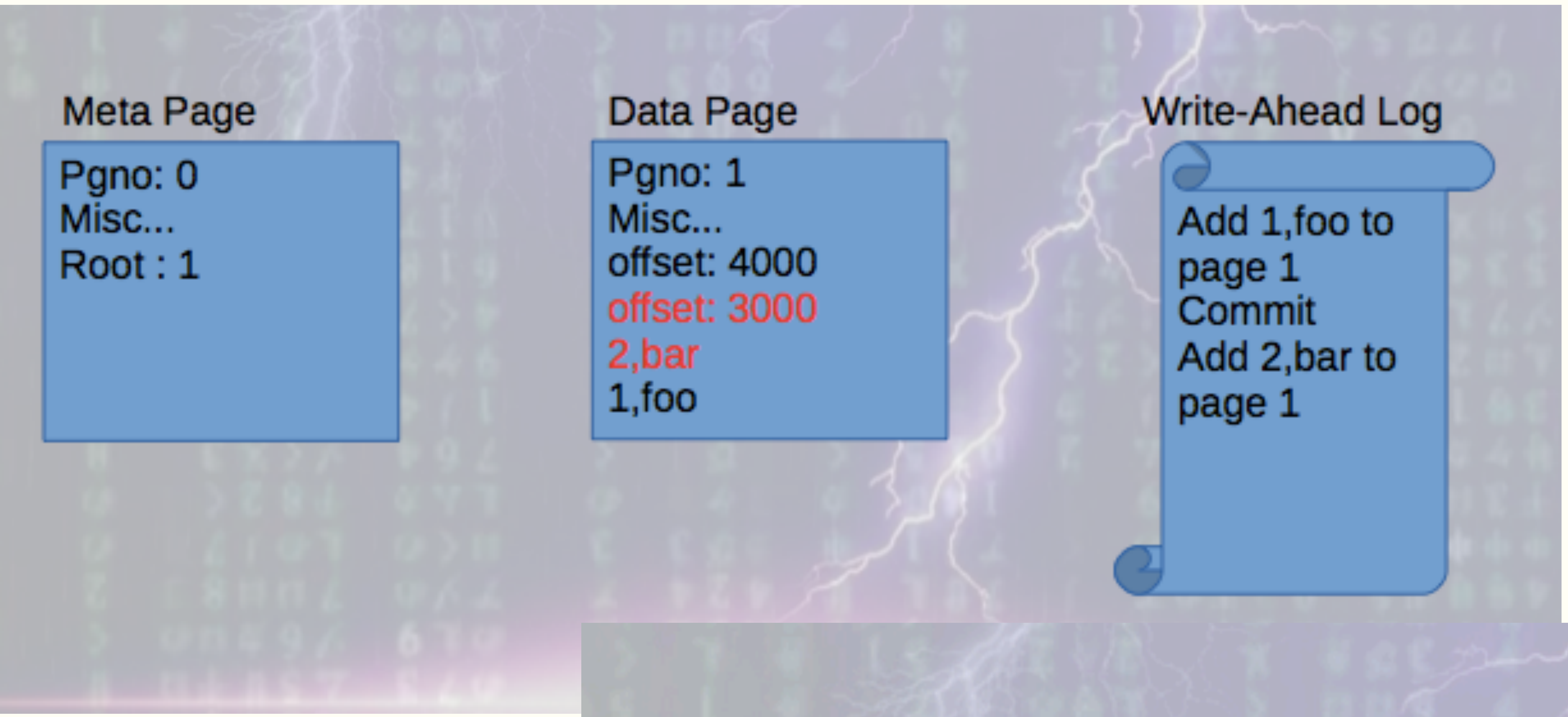


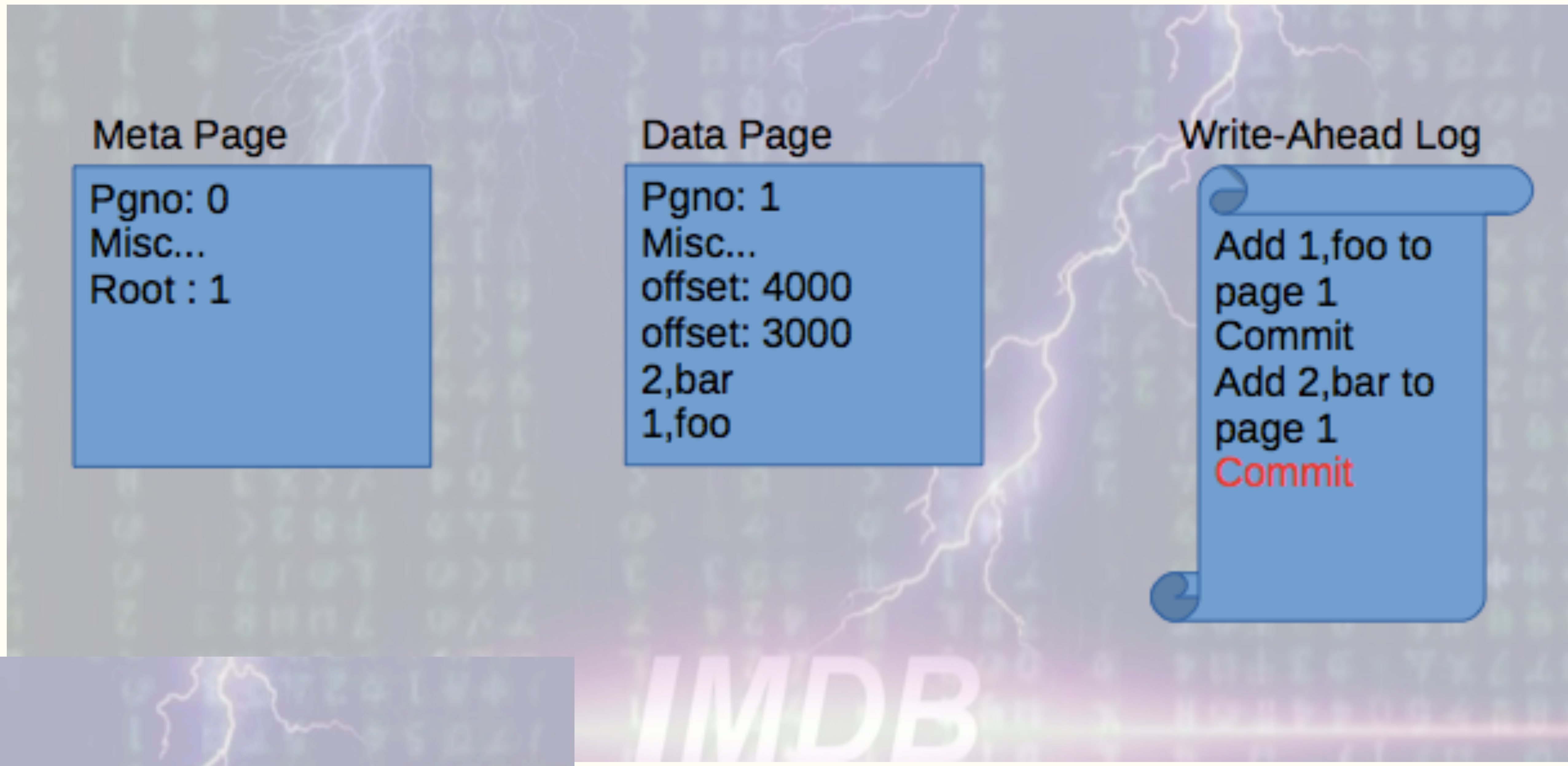
Btrees

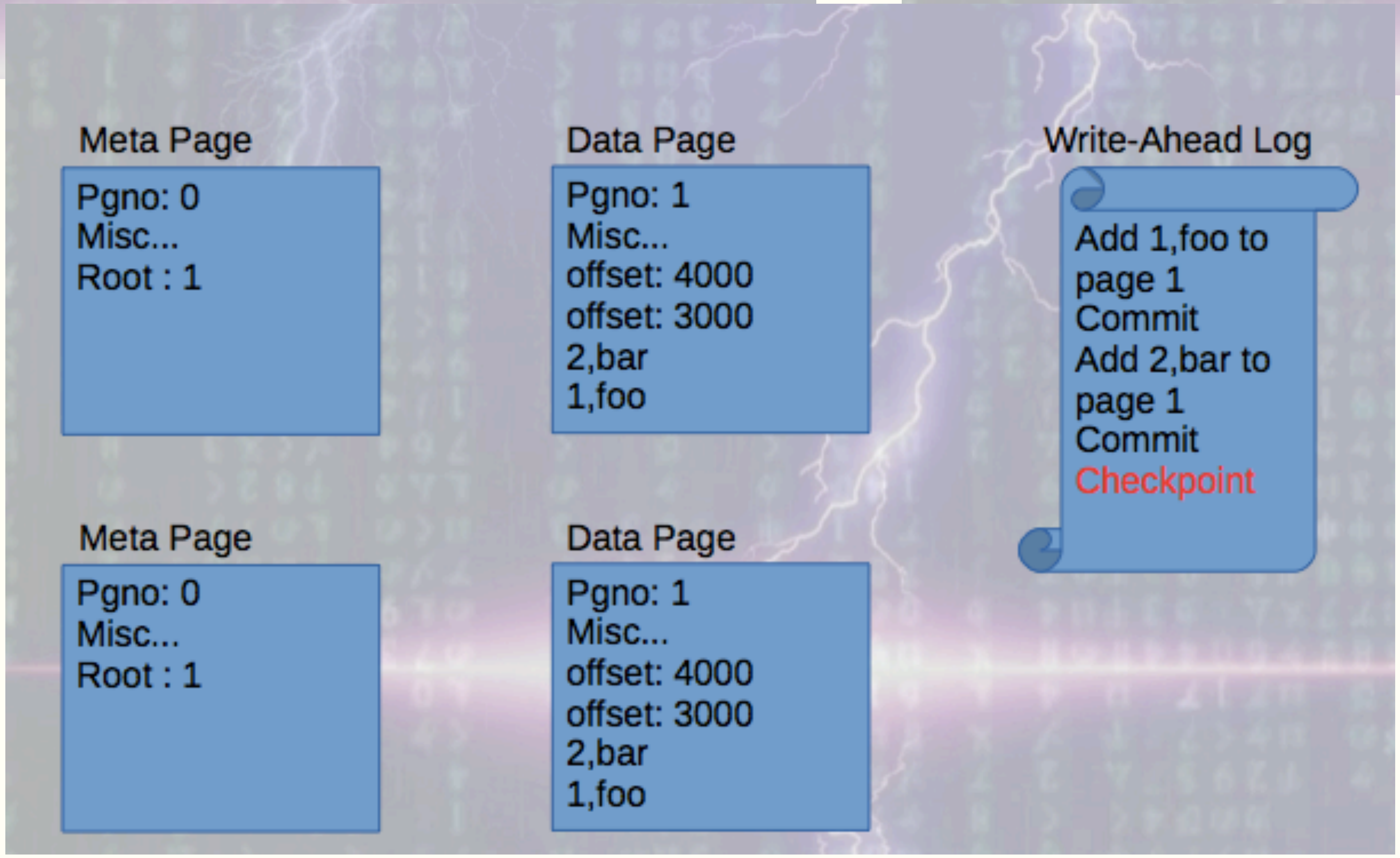
Univ.AI

BTrees

- (from https://loveforprogramming.quora.com/Memory-locality-the-magic-of-B-Trees): ”A linked sorted distributed range array with predefined sub array size which allows searches, sequential access, insertions, and deletions in logarithmic time. ”
- It is a generalization of a binary tree, but the branching factor is much higher, and the depth thus smaller
- btrees break database into pages, and read-or-write one page at a time. A page is about 4k in size (see https://www.tutorialspoint.com/operating_system/os_virtual_memory.htm)
- Leaf pages contain all the values and may represent a clustered index
- The pointers in a btree are disk based pointers
- Both splits and writing in-place are dangerous, so its normal for b-tree implementations to have a WAL, or write ahead log (such a log can also be used to manage transactions). Every operation on the btree is appended to this log file.


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.

3. Transactions and ACID

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.

Univ.AI

- 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.

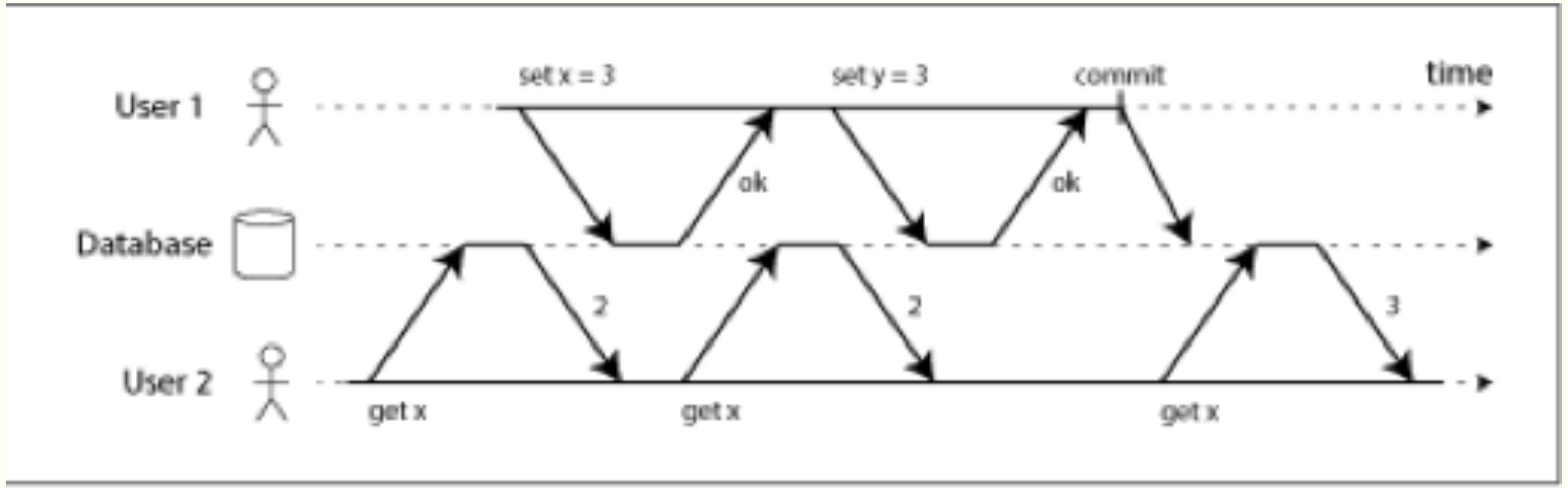
- Clearly, the notions of isolation are really the notions of concurrency: these issues will also occur when 2 programs access any data, in memory or in a database. In both cases locks and other ideas must be used to make sure that there is only one mutator at a time, and that an object is not exposed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

Read Committed Isolation Level

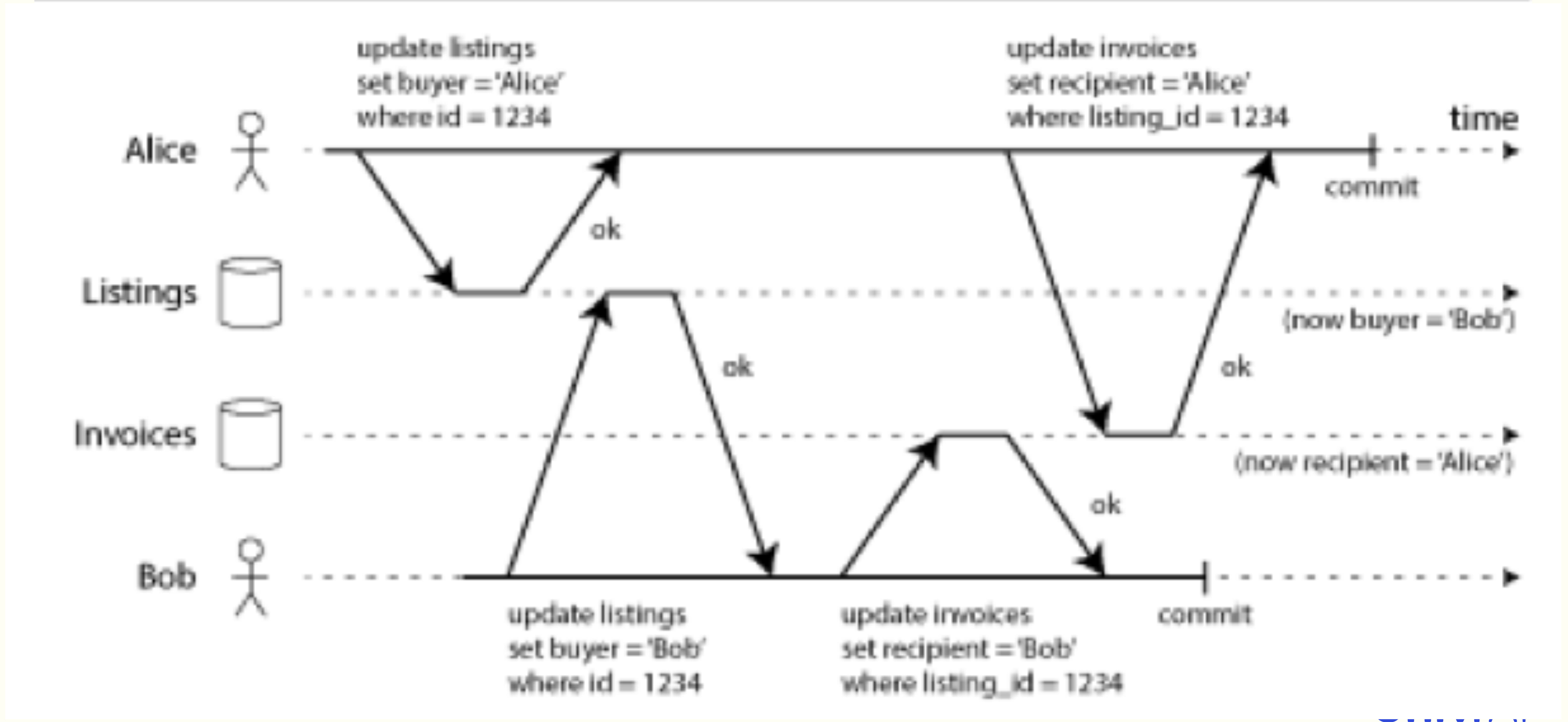
- No dirty reads and no dirty writes.!

- use row-level locks. Another writer must wait until the lock is given up, and the reader will read the value before the lock was set as long as the write transaction is in progress.

No Dirty Reads



A dirty write:



Read Committed

- Thus, in Read Comitted mode, in essence we need to remember the pre-transaction value, ie the old committed value and the also the new value being set by the transaction which holds the current write lock.

- **While this transaction runs, any other thread/transaction sees the old value**. Once this one commits, then the values read by the other transaction are switched.

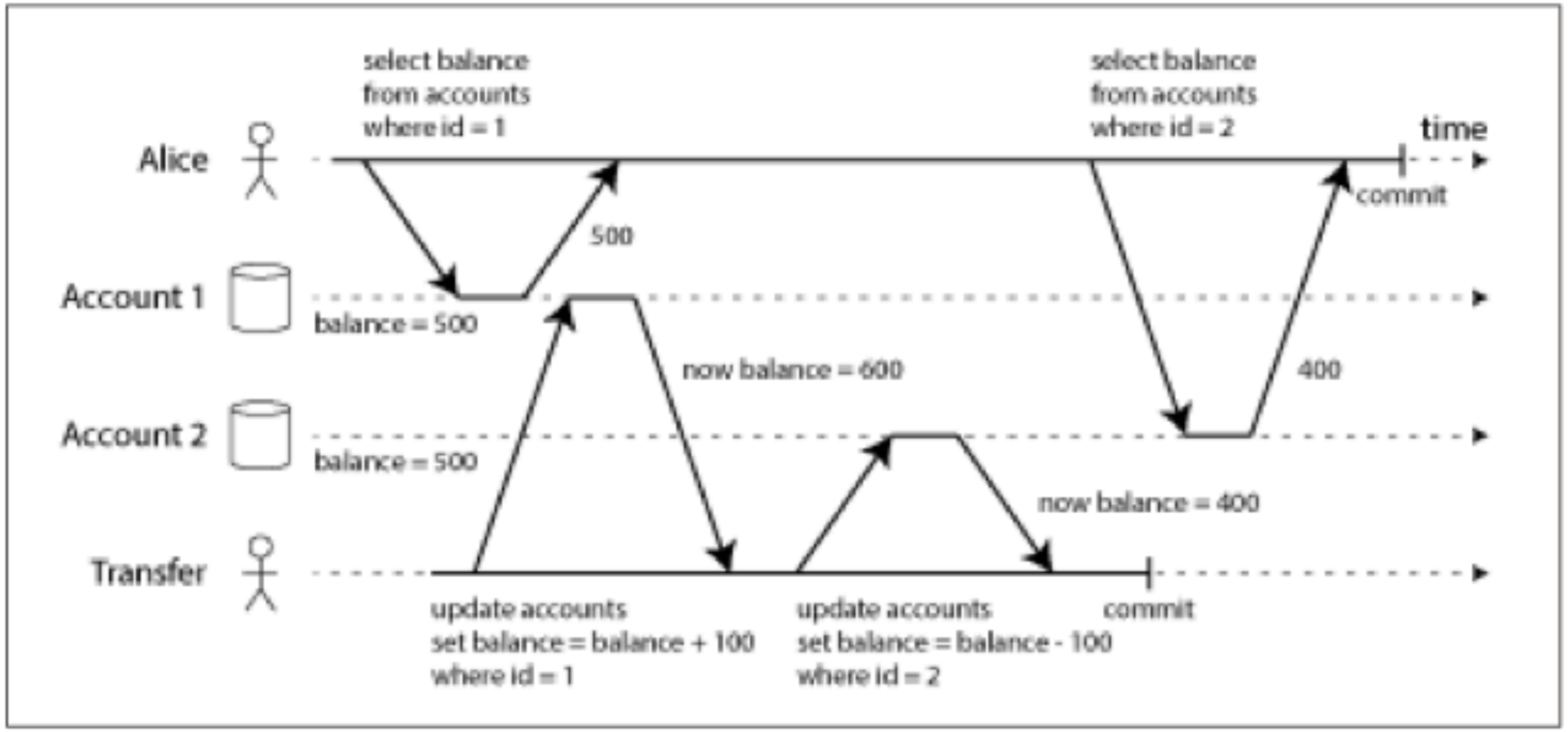
- Read Comitted is almost always implemented: its the default in Oracle, Postgres, SQL Server, etc

- WHAT COULD GO WRONG?

The Read Skew Problem and Snapshot Isolation

- Non-repeatable reads: A client sees different parts of the database at different points in time.

- In this use case it might not be an issue, but for backups and analytics you dont want to read while a transaction is going on



select balance from accounts where id = 1
select balance from accounts where id = 2
time
Alice
commit
500
Account 1
balance = 500
now balance = 600
400
Account 2
balance = 500
now balance = 400
Transfer
update accounts set balance = balance + 100 where id = 1
update accounts set balance = balance - 100 where id = 2
commit

The fix for this would seem to be clear: if both the reads happen in one transaction, one must make sure that because this transaction happened before the write, it sees the older version of the accounts (500, 500).

Univ.AI

MVCC

- These thoughts form the basis of Multi-Version concurrency control, or MVCC, also known as snapshot isolation, which allows a transaction to read from a consistent snapshot at one point in time. This is needed for analytic queries testing integrity, backups, etc. Snapshot isolation is a useful isolation level, especially for read-only transactions.

- The key principle: readers never block writers, and writers never block readers

- Snapshot isolation is supported by postgres, MySQL with the InnoDB storage engine, Oracle, SQL Server, and more, but may not be the default, as its less performant than read-committed isolation.

- Indexes in a MVCC database are hard to get right…and affect performance as they need updating. One way to deal with this is COW (copy on write).

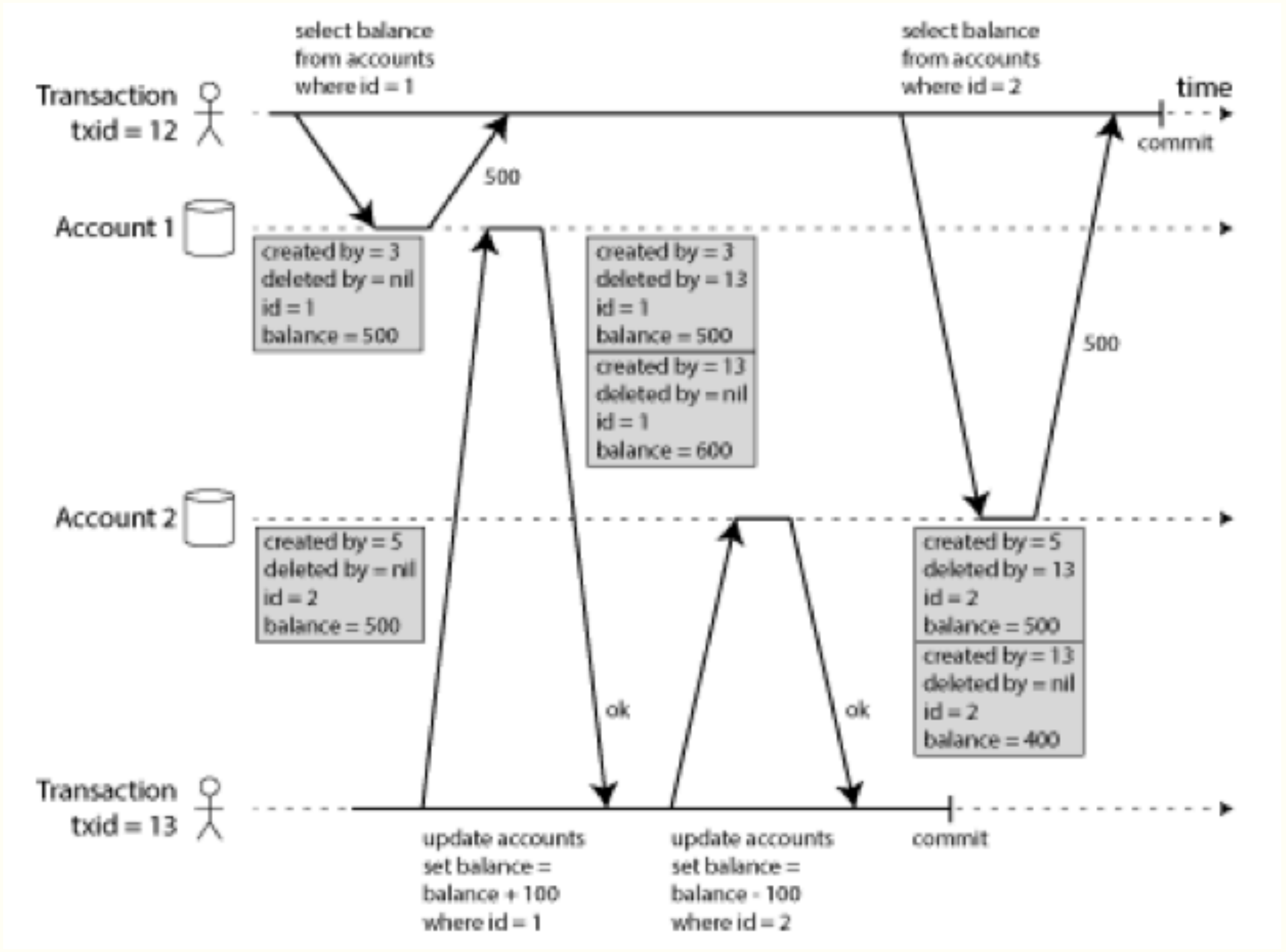
Rules for MVCC

1. At the start of each transaction, the database makes a list of all the other transations which are in progress (not yet committed or aborted) at that time. Any writes made by one of those transactions are ignored, even if the transaction subsequently commits.

2. Any writes made by aborted transactions are ignored.

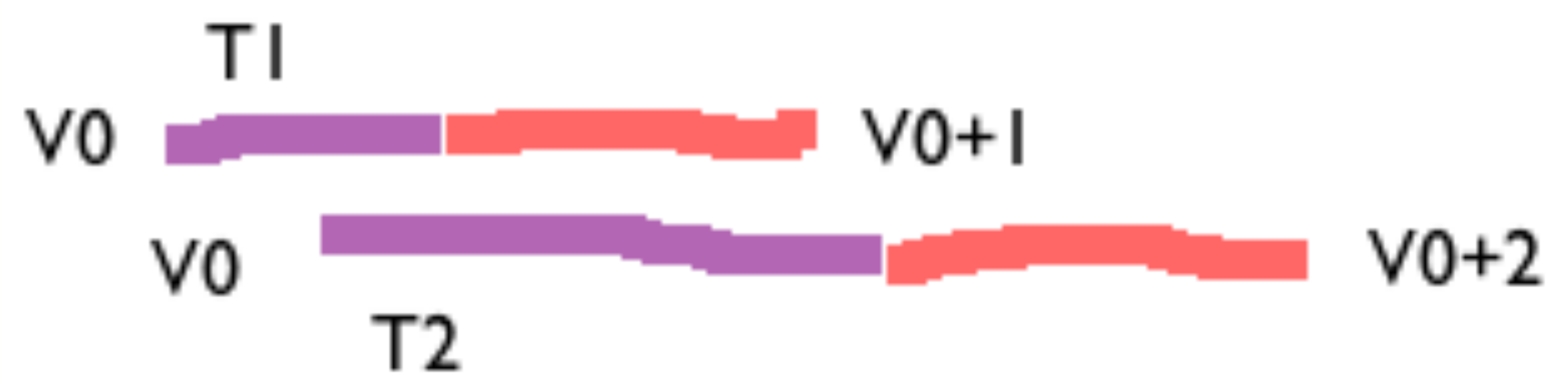
3. Any writes made by transactions with a later transaction ID (i.e. which started after the current transaction started) are ignored, regardless of whether that transaction has committed.

4. All other writes are visible to the application's queries.



Univ.AI

The Lost Update Problem



- Consider an upsert, where we read a value, do something with it, like the balance change above, and then write it. Consider two such upserting transactions T1 and T2. Because we lock writes, a transaction T1 might read a value V0, and then transaction T2 starting when T1 is on can only read value V0. Since T2 does not (yet) have lock, the code in a `get` uses the old value. This is good as the transaction T1 is still running.

- But here is the problem: T1 will create a new value, say with V0+1. It then relinquishes the lock. Now T2 comes gets lock, operates on V0, and makes it V0+2.

- Say V0 was 5 to start with. The notion of the process might have been 5->6->8. But we wont do that. Our latest value will be 7.

Univ.AI

Fixing The Lost Update Problem

- The fix to this is atomic updates/upserts. The `UPDATE` command in sql does just that, and if we build transactional facility into our database to do multiple gets, there is no reason not to extend it to an upsert. The entire upsert would grab the lock, and once that happens this would effectively serialize T1 and T2. Atomic updates are usually thus built by taking the lock when the object is read, or by forcing all atomic operations to be on one given thread.

- So thus it seems we can add a transaction manager, have an explicit notion of read, write, and upsert transactions, and have a reasonable snapshot isolated database.

- There are other techniques as well. But even this does not solve some problems...

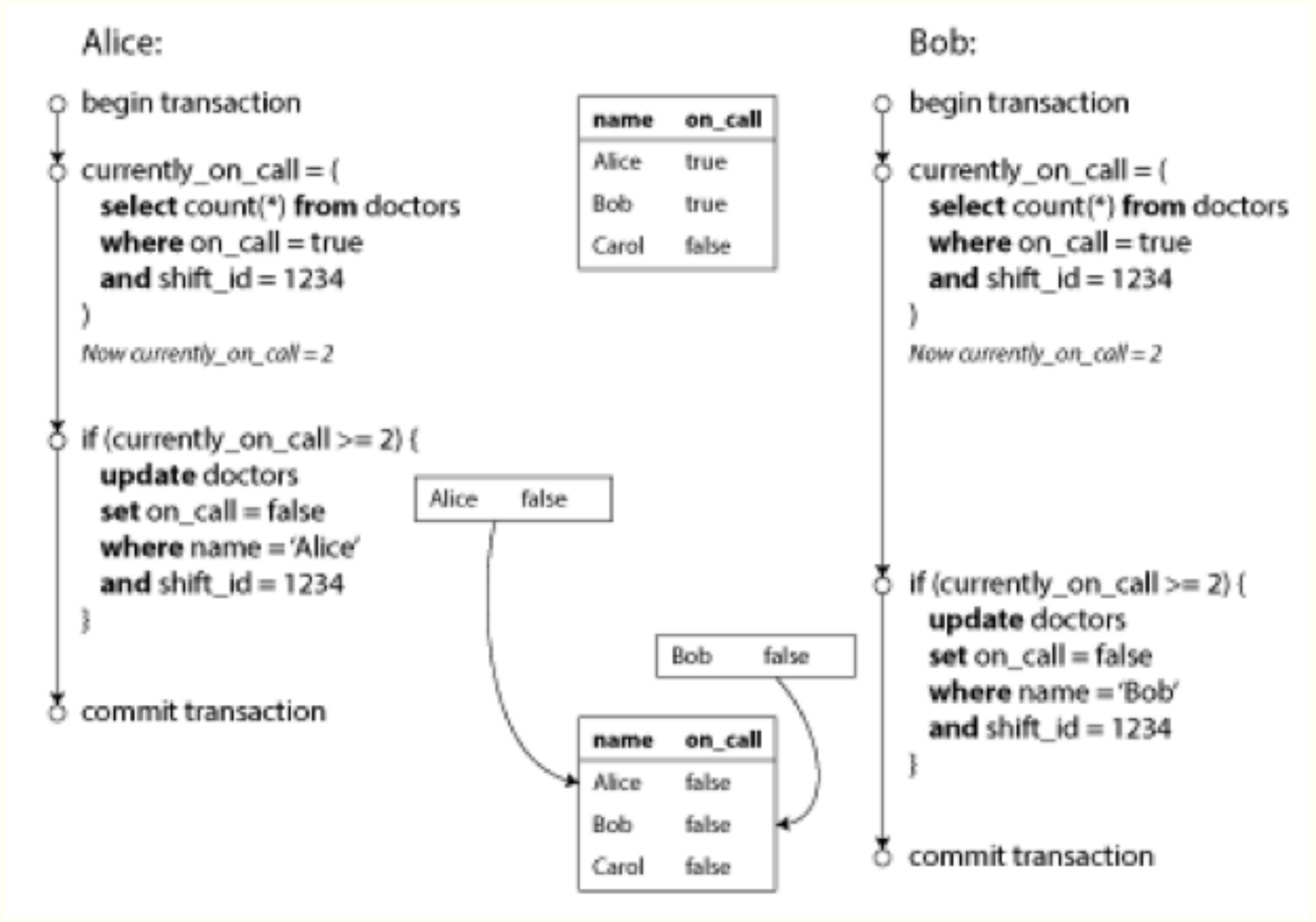
Univ.AI

Write Skew

- Both Alice and Bob are last docs on a shift. They both want to leave. So they both go to their web portal and click on the "check-out" button. The app is using snapshot isolation. Since they both click at roughly the same time, before any one's check-out transaction has completed, the both "get" a count of two doctors on call and their transactions are allowed to proceed, leaving no docs. oops.

- This anomaly happens as the transactions here are updating different rows.

- Another example is conflicting meetings being booked in a room at the same time.



Alice:

```
begin transaction

currently_on_call = (
  select count(*) from doctors
  where on_call = true
  and shift_id = 1234
)
Now currently_on_call = 2

if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Alice'
  and shift_id = 1234
}

commit transaction
```

Bob:

```
begin transaction

currently_on_call = (
  select count(*) from doctors
  where on_call = true
  and shift_id = 1234
)
Now currently_on_call = 2

if (currently_on_call >= 2) {
  update doctors
  set on_call = false
  where name = 'Bob'
  and shift_id = 1234
}

commit transaction
```

name	on_call
Alice	true
Bob	true
Carol	false

Alice	false

Bob	false

name	on_call
Alice	false
Bob	false
Carol	false

Univ.AI

Write Skew and Phantoms

- The pattern here is that there is a SELECT query that two transactions do that would have returned different results based on which a write would have happened or not. But these two transactions were touching two different rows or parts of the dbase and were thus allowed under MVCC. You can think of write-skew as a generalization of lost-update.

- This effect, where a write in one transaction changes the result of a search query in another transaction, is called a phantom. Snapshot isolation avoids phantoms in read-only queries, but in read-write transactions like the examples we discussed, phantoms can lead to particularly tricky cases of write skew.

- One could solve these problems in multiple ways. The most general solution is to use serialized isolation.

Serialized isolation with 2 phase locking

- if a transaction wants to read an object, it must first acquire the lock in shared mode. Several transactions are allowed to hold the lock in shared mode simultaneously, but if another transaction already has an exclusive lock on the object, the transaction must wait.

- If a transaction wants to write to an object, it must first acquire the lock in exclusive mode. No other transaction may hold the lock at the same time (neither in shared nor in exclusive mode), so if there is any existing lock on the object, the transaction must wait.

- If a transaction first reads and then writes an object, it may upgrade its shared lock to an exclusive lock. The upgrade works the same as getting an exclusive lock directly.

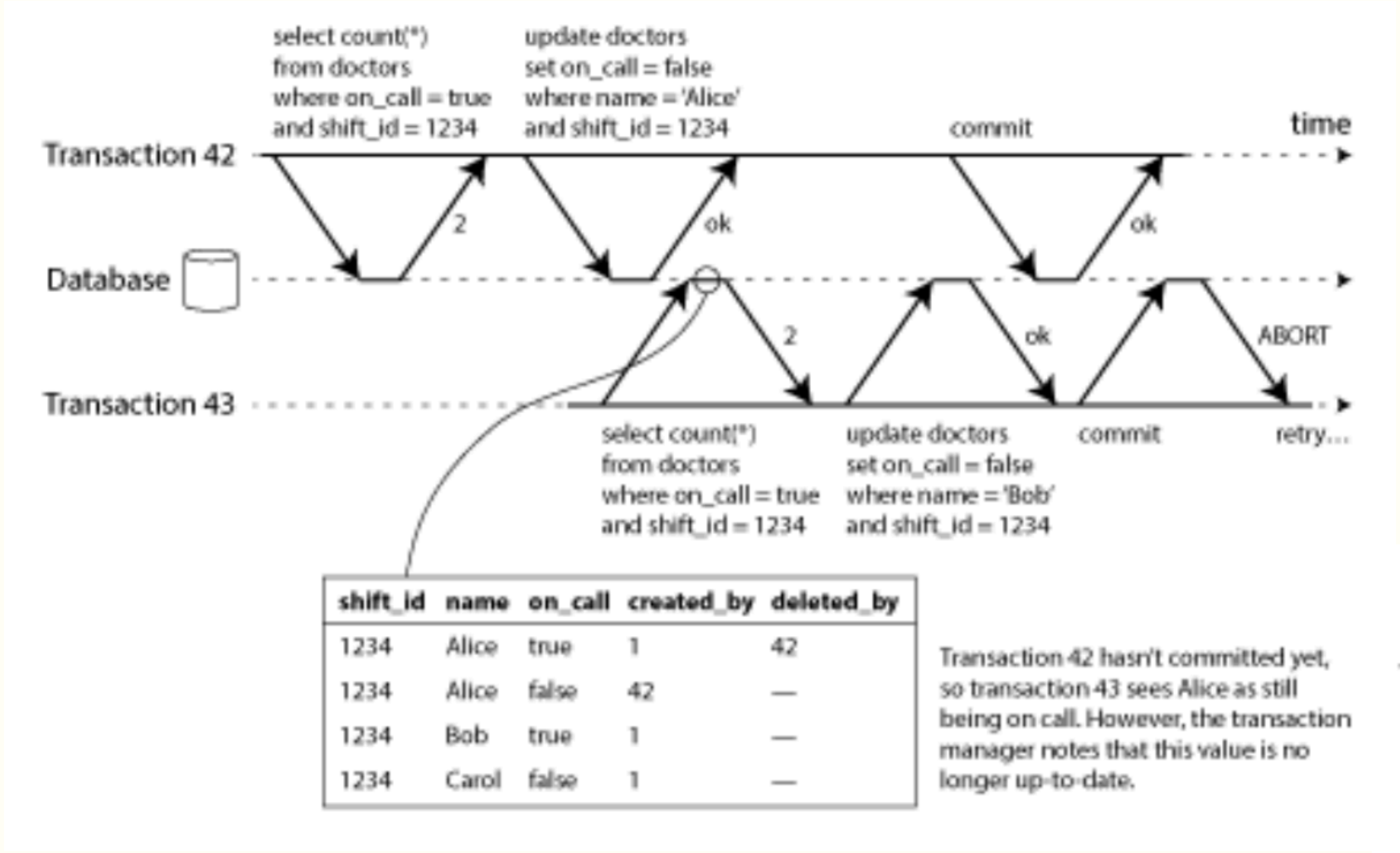
- After a transaction has acquired the lock, it must continue to hold the lock until the end of the transaction (commit or abort). This is where the name "two- phase" comes from: the first phase (while the transaction is executing) is when the locks are acquired, and the second phase (at the end of the transaction) is when all the locks are released

- You can also completely serialize transactions onto one thread. Both 2-phase locking and serial transactions slow things down.

Serializable Snapshot Isolation

- optimistic concurrency control technique. Optimistic in this context means that instead of blocking if something poten- tially dangerous happens, transactions continue anyway, in the hope that everything will turn out alright.

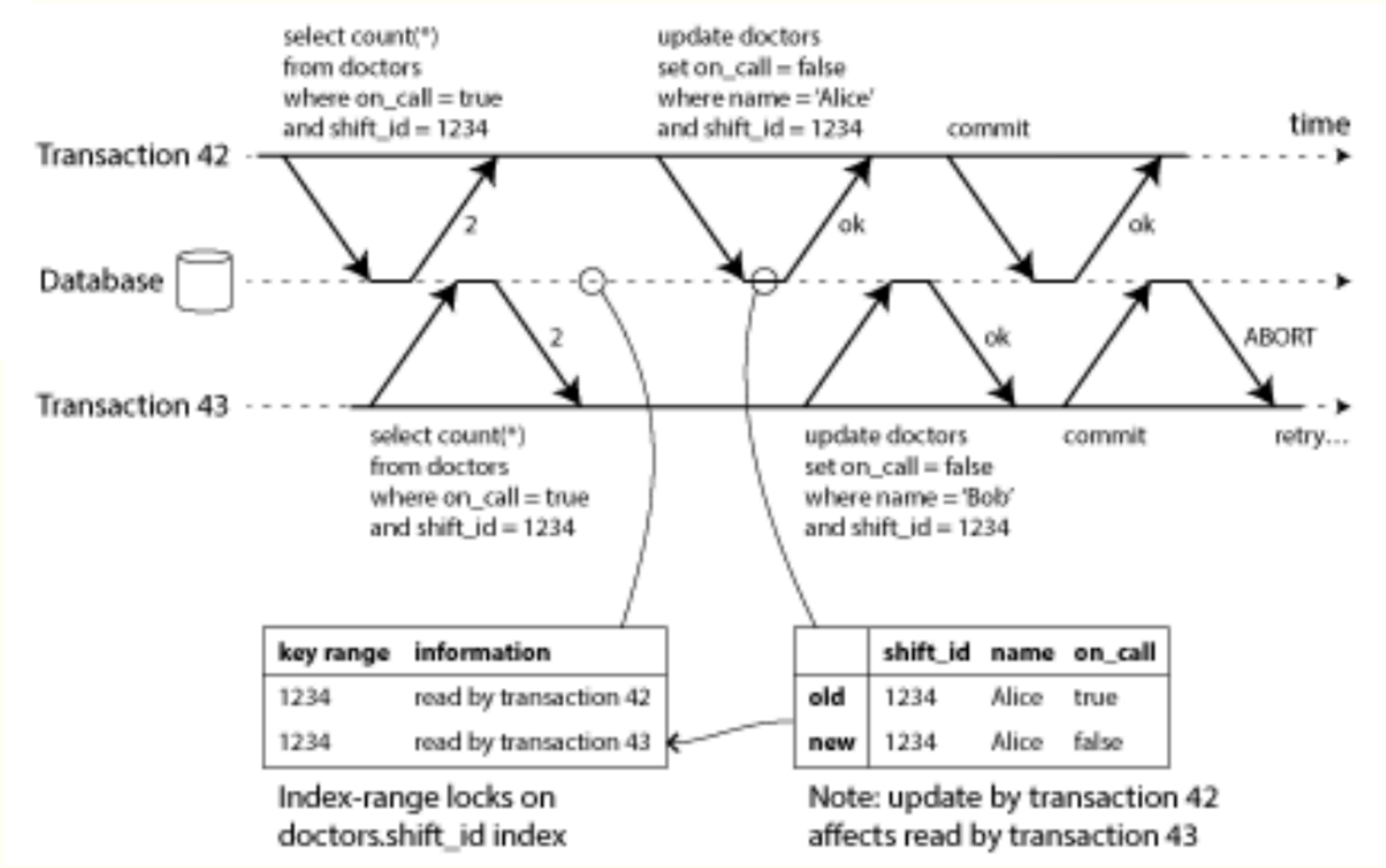
- When a transaction wants to commit, the database checks whether anything bad happened (i.e. whether isolation was violated); if so, the trans- action is aborted and has to be retried.



shift_id	name	on_call	created_by	deleted_by
1234	Alice	true	1	42
1234	Alice	false	42	—
1234	Bob	true	1	—
1234	Carol	false	1	—

Transaction 42 hasn't committed yet, so transaction 43 sees Alice as still being on call. However, the transaction manager notes that this value is no longer up-to-date.

uncomitted write occurred before read



key range	information
1234	read by transaction 42
1234	read by transaction 43

Index-range locks on doctors.shift_id index

	shift_id	name	on_call
old	1234	Alice	true
new	1234	Alice	false

Note: update by transaction 42 affects read by transaction 43

write occurs after the read

Univ.AI