MySQL

Let’s begin at 19:05 Minsk time

MySQL is an open-source relational database management system (RDBMS)

- **SQL - structured query language**
- **Data integrity**
- **Fault tolerance**

341 systems in ranking, December 2018

Rank			DBMS	Database Model	Score		
Dec 2018	Nov 2018	Dec 2017			Dec 2018	Nov 2018	Dec 2017
1.	1.	1.	Oracle	Relational DBMS	1283.22	-17.89	-58.32
2.	2.	2.	MySQL	Relational DBMS	1161.25	+1.36	-156.82
3.	3.	3.	Microsoft SQL Server	Relational DBMS	1040.34	-11.21	-132.14
4.	4.	4.	PostgreSQL	Relational DBMS	460.64	+20.39	+75.21
5.	5.	5.	MongoDB	Document store	378.62	+9.14	+47.85
6.	6.	6.	IBM Db2	Relational DBMS	180.75	+0.87	-8.83
7.	7.	↑ 8.	Redis	Key-value store	146.83	+2.66	+23.59
8.	8.	↑ 10.	Elasticsearch	Search engine	144.70	+1.24	+24.92
9.	9.	↓ 7.	Microsoft Access	Relational DBMS	139.51	+1.08	+13.63
10.	10.	↑ 11.	SQLite	Relational DBMS	123.02	+0.31	+7.82

Rank			DBMS	Database Model	Score		
Oct 2020	Sep 2020	Oct 2019			Oct 2020	Sep 2020	Oct 2019
1.	1.	1.	Oracle	Relational, Multi-model	1368.77	-0.59	+12.89
2.	2.	2.	MySQL	Relational, Multi-model	1256.38	-7.87	-26.69
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	1043.12	-19.64	-51.60
4.	4.	4.	PostgreSQL	Relational, Multi-model	542.40	+0.12	+58.49
5.	5.	5.	MongoDB	Document, Multi-model	448.02	+1.54	+35.93
6.	6.	6.	IBM Db2	Relational, Multi-model	161.90	+0.66	-8.87
7.	↑ 8.	7.	Elasticsearch	Search engine, Multi-model	153.84	+3.35	+3.67
8.	↓ 7.	8.	Redis	Key-value, Multi-model	153.28	+1.43	+10.37
9.	9.	↑ 11.	SQLite	Relational	125.43	-1.25	+2.80
10.	10.	10.	Cassandra	Wide column	119.10	-0.08	-4.12

383 systems in ranking, February 2022

Rank			DBMS	Database Model	Score		
Feb 2022	Jan 2022	Feb 2021			Feb 2022	Jan 2022	Feb 2021
1.	1.	1.	Oracle	Relational, Multi-model	1256.83	-10.05	-59.84
2.	2.	2.	MySQL	Relational, Multi-model	1214.68	+8.63	-28.69
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	949.05	+4.24	-73.88
4.	4.	4.	PostgreSQL	Relational, Multi-model	609.38	+2.83	+58.42
5.	5.	5.	MongoDB	Document, Multi-model	488.64	+0.07	+29.69
6.	6.	↑ 7.	Redis	Key-value, Multi-model	175.80	-2.18	+23.23
7.	7.	↓ 6.	IBM Db2	Relational, Multi-model	162.88	-1.32	+5.26
8.	8.	8.	Elasticsearch	Search engine, Multi-model	162.29	+1.54	+11.29
9.	9.	↑ 11.	Microsoft Access	Relational	131.26	+2.31	+17.09
10.	10.	↓ 9.	SQLite	Relational	128.37	+0.94	+5.20

Rank			DBMS	Database Model	Score		
Jan 2025	Dec 2024	Jan 2024			Jan 2025	Dec 2024	Jan 2024
1.	1.	1.	Oracle	Relational, Multi-model	1258.76	-5.03	+11.27
2.	2.	2.	MySQL	Relational, Multi-model	998.15	-5.61	-125.31
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	798.55	-7.14	-78.05
4.	4.	4.	PostgreSQL	Relational, Multi-model	663.41	-2.97	+14.45
5.	5.	5.	MongoDB	Document, Multi-model	402.50	+2.12	-14.98
6.	↑ 7.	↑ 9.	Snowflake	Relational	153.90	+6.54	+27.98
7.	↓ 6.	↓ 6.	Redis	Key-value, Multi-model	153.36	+3.08	-6.03
8.	8.	↓ 7.	Elasticsearch	Multi-model	134.92	+2.60	-1.15
9.	9.	↓ 8.	IBM Db2	Relational, Multi-model	122.97	+0.19	-9.43
10.	10.	↑ 11.	SQLite	Relational	106.69	+4.97	-8.51

Ranking > Method

Method of calculating the scores of the DB-Engines Ranking

The DB-Engines Ranking is a list of database management systems ranked by their current popularity. We measure the popularity of a system by using the following parameters:

- **Number of mentions of the system on websites**, measured as number of results in search engines queries. At the moment, we use Google and Bing for this measurement. In order to count only relevant results, we are searching for <system name> together with the term database, e.g. "Oracle" and "database".
- **General interest in the system.** For this measurement, we use the frequency of searches in Google Trends.
- **Frequency of technical discussions about the system.** We use the number of related questions and the number of interested users on the well-known IT-related Q&A sites Stack Overflow and DBA Stack Exchange.
- **Number of job offers, in which the system is mentioned.** We use the number of offers on the leading job search engines Indeed and Simply Hired.
- **Number of profiles in professional networks, in which the system is mentioned.** We use the internationally most popular professional network LinkedIn.
- **Relevance in social networks.** We count the number of Twitter tweets, in which the system is mentioned.

MySQL: GitHub, US Navy, NASA, Tesla, Netflix, WeChat, Facebook, Zendesk, Twitter, Zappos, YouTube, Spotify.

Oracle: Bauerfeind AG, CAIRN India, Capcom Co., ChevronTexaco, Coca-Cola FEMSA, COOP Switzerland, ENEL, Heidelberger Druck, MTU Aero Engines, National Foods Australia, Spire Healthcare, Stadtwerke München, Swarovski, Tyson Foods, TVS Motor Company, Vilene.

- Data types;
- Architecture;
- Indexes;
- ACID

Numeric Data Types

- **TINYINT**: from -127 to 128, takes 1 byte
- **BOOL (BOOLEAN)**: `TINYINT(1)`.
- **TINYINT UNSIGNED**: from 0 to 255, takes 1 byte
- **SMALLINT**: from -32768 to 32767, takes 2 bytes
- **SMALLINT UNSIGNED**: from 0 to 65535, takes 2 bytes
- **MEDIUMINT**: from -8388608 to 8388607, takes 3 bytes
- **MEDIUMINT UNSIGNED**: from 0 to 16777215, takes 3 bytes
- **INT**: from -2147483648 to 2147483647, takes 4 bytes
- **INT UNSIGNED**: from 0 to 4294967295, takes 4 bytes
- **BIGINT**: from -9 223 372 036 854 775 808 to 9 223 372 036 854 775 807, takes 8 bytes
- **BIGINT UNSIGNED**: from 0 to 18 446 744 073 709 551 615, takes 8 bytes

Numeric Data Types

- **DECIMAL**: Fixed-precision number. `DECIMAL(precision, scale)`. 1 < `precision < 65`
- **FLOAT**: $-3.4028 * 10^{38}$ to $3.4028 * 10^{38}$, takes 4 bytes `FLOAT(M,D)`
- **DOUBLE**: from $-1.7976 * 10^{308}$ to $1.7976 * 10^{308}$, takes 8 bytes. `DOUBLE(M,D)`

Data types for a work with date and time

- **DATE**: stores dates from JAN 1 1000 to DEC 31 9999. By default it stores data using yyyy-mm-dd format. Takes 3 bytes.
- **TIME**: stores time from -838:59:59 to 838:59:59. By default it stores data using “hh:mm:ss" format. Takes 3 bytes.
- **DATETIME**: from "1000-01-01 00:00:00" to "9999-12-31 23:59:59". By default it stores data using "yyyy-mm-dd hh:mm:ss” format. Takes 8 bytes
- **TIMESTAMP**: from "1970-01-01 00:00:01" UTC to "2038-01-19 03:14:07" UTC. Takes 4 bytes
- **YEAR**: stores a year as 4 digits. Takes 1 byte.

String Types

Can build index

- **CHAR**: fixed length string
- **VARCHAR**: variable length string

Can build index with limited length

- **TINYTEXT**: text up to 255 bytes.
- **TEXT**: text up to 65 KB.
- **MEDIUMTEXT**: text up to 16 MB
- **LARGETEXT**: text up to GB

Union Types

- **ENUM**: stores single value from the list of allowed values. Takes 1-2 bytes

```
CREATE TABLE tickets (
     id INT PRIMARY KEY AUTO_INCREMENT,
     title VARCHAR(255) NOT NULL,
     priority ENUM('Low', 'Medium', 'High') NOT NULL
);
```

```
INSERT INTO tickets(title, priority) VALUES('Scan virus for computer A', 'High');

INSERT INTO tickets(title, priority) VALUES('Upgrade Windows OS for all computers', 1);
```

	id	title	priority
1	1	Scan virus for computer A	High
2	2	Upgrade Windows OS for all computers	Low

Union Types

- **SET**: stores multiple values from the list of allowed values (up to 64 possible values). Takes 1-8 bytes.

```
CREATE TABLE tickets2 (
     id INT PRIMARY KEY AUTO_INCREMENT,
     title VARCHAR(255) NOT NULL,
     priority SET('2^0', '2^1', '2^2') NOT NULL
);
```

```
INSERT INTO tickets2(title, priority) VALUES('Show example', ‘2^0,2^1');
INSERT INTO tickets2(title, priority) VALUES('Show example 2', 5);
```

	id	title	priority
1	1	Show example	2^0,2^1
2	2	Show example 2	2^0,2^2

Binary data types

- **TINYBLOB**: binary text up to 255 bytes.
- **BLOB**: binary text up to 65 KB.
- **MEDIUMBLOB**: binary text up to 16 MB
- **LARGEBLOB**: binary text up to 4 GB

JSON data type

Native JSON support which includes

1. Automatic JSON validation
2. Optimized storage
3. Partial update using
    1. JSON_SET
    2. JSON_INSERT
    3. JSON_REPLACE
    4. …

ORM

```
// Create a new user
const jane = await User.create({ firstName: "Jane", lastName: "Doe" });
console.log("Jane's auto-generated ID:", jane.id);
```

```
const { Op } = require("sequelize");
Post.findAll({
  where: {
    authorId: {
      [Op.eq]: 2
    }
  }
});
// SELECT * FROM post WHERE authorId = 2;
```

```
// Change everyone without a last name to "Doe"
await User.update({ lastName: "Doe" }, {
  where: {
    lastName: null
  }
});
```

```sql
CREATE TABLE Users
(
    id INT UNSIGNED NOT NULL PRIMARY KEY AUTO_INCREMENT,
    user_id INT  NOT NULL,
    user_name CHAR(64) NOT NULL
);
```

```sql
DROP TABLE Users;
```

```sql
ALTER TABLE Users
  MODIFY COLUMN user_name VARCHAR(64)  NOT NULL,
  ADD COLUMN group_id TINYINT(3) DEFAULT NULL AFTER user_id,
  ADD INDEX name (user_name),
  DROP COLUMN surname,
  ADD CONSTRAINT FK_Group  FOREIGN KEY (group_id) REFERENCES Groups (group_id);
```

SELECT * FROM Users (WHERE user_id = 1);

UPDATE Users SET user_name = 'Иванов И.И.' WHERE user_id = 1;

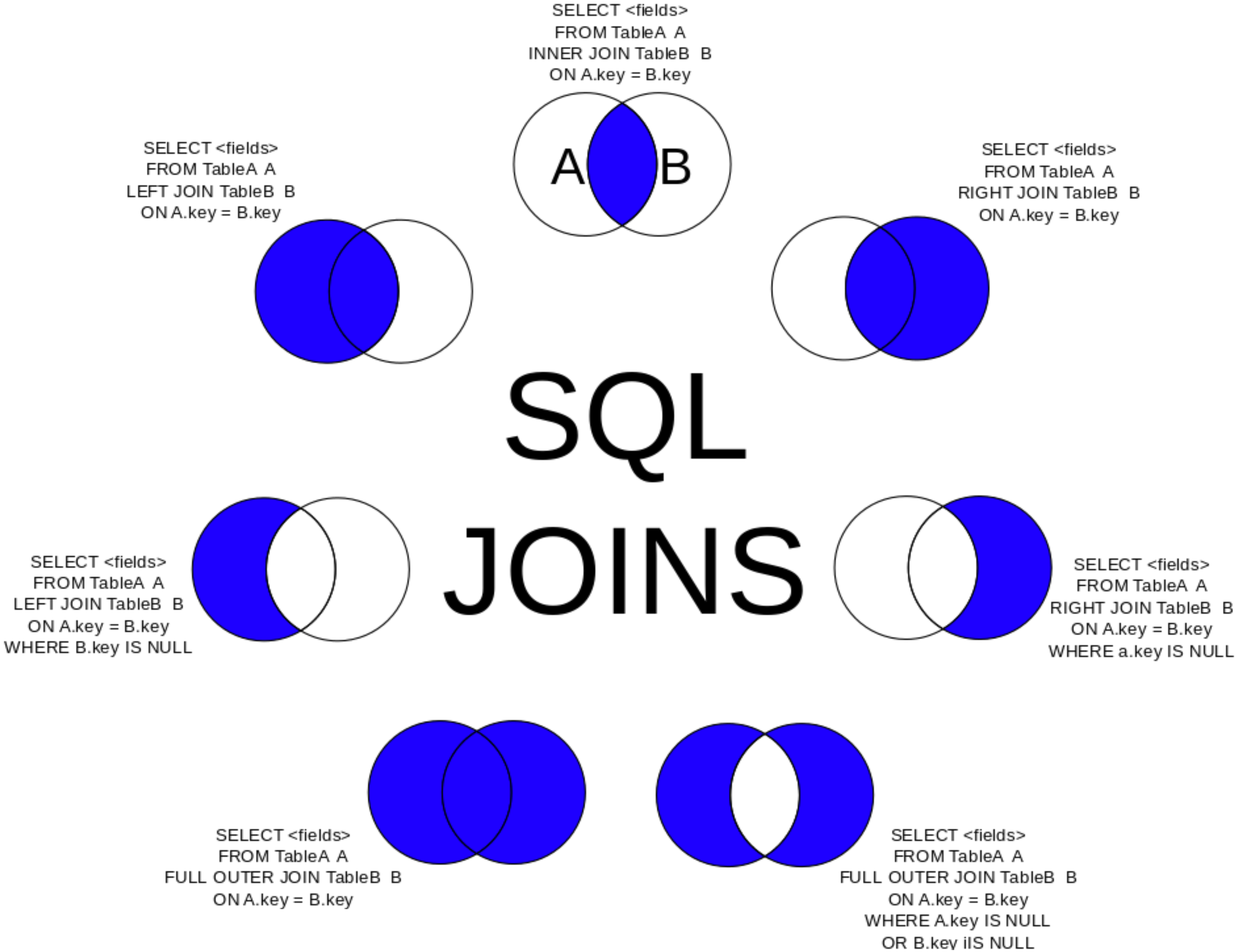
INSERT INTO Users (user_id, group_id, user_name) VALUES (2, 1, 'Петров П.П.');

INSERT INTO Users VALUES (3, 1, ‘Петрович П.П.'), (4, NULL, 'Сидоров С.С.');

DELETE FROM Users WHERE user_id = 1;

```sql
SELECT * FROM Users a
   INNER JOIN  Groups g
       ON a.group_id = g.group_id
   WHERE user_id = 1;
```

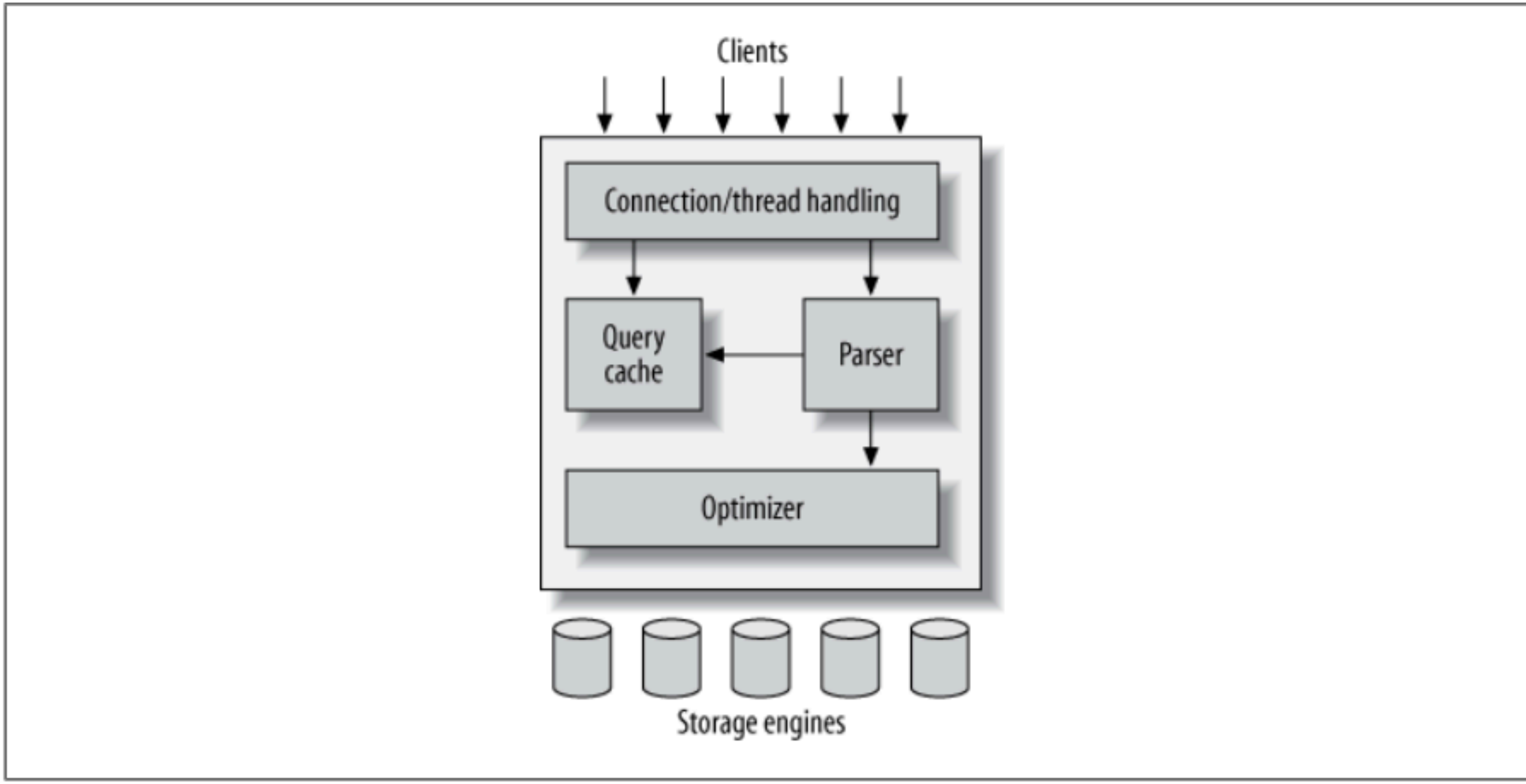
```sql
SELECT * FROM Users a
   LEFT JOIN  Groups g
       ON a.group_id = g.group_id
   WHERE user_id = 1;
```


SQL
JOINS
SELECT <fields>
FROM TableA A
INNER JOIN TableB B
ON A.key = B.key
A
B
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
SELECT <fields>
FROM TableA A
LEFT JOIN TableB B
ON A.key = B.key
WHERE B.key IS NULL
SELECT <fields>
FROM TableA A
RIGHT JOIN TableB B
ON A.key = B.key
WHERE a.key IS NULL
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
SELECT <fields>
FROM TableA A
FULL OUTER JOIN TableB B
ON A.key = B.key
WHERE A.key IS NULL
OR B.key iIS NULL

List groups which have more than 2 users excluding test users

```
SELECT g.group_name, COUNT(*) as cnt
   FROM Users u
   INNER JOIN  Groups g
      ON u.group_id = g.id
   WHERE u.is_test=0
   GROUP BY g.id
   HAVING cnt > 2;
```

Architecture



Storage engines

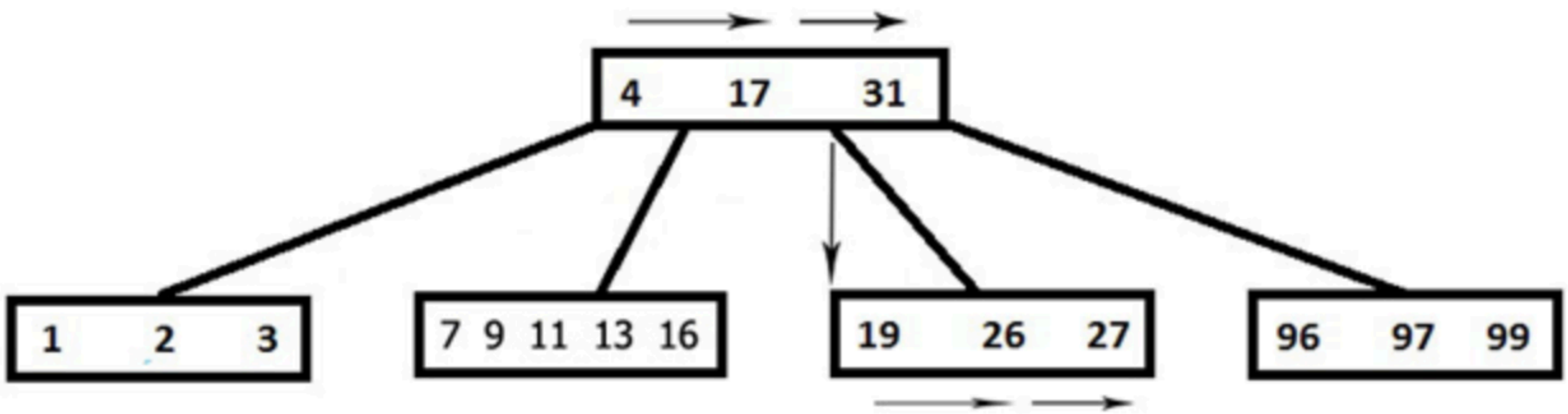
- **MyISAM**
- **Memory**
- **CSV**
- **Archive**
- **Blackhole**

InnoDB

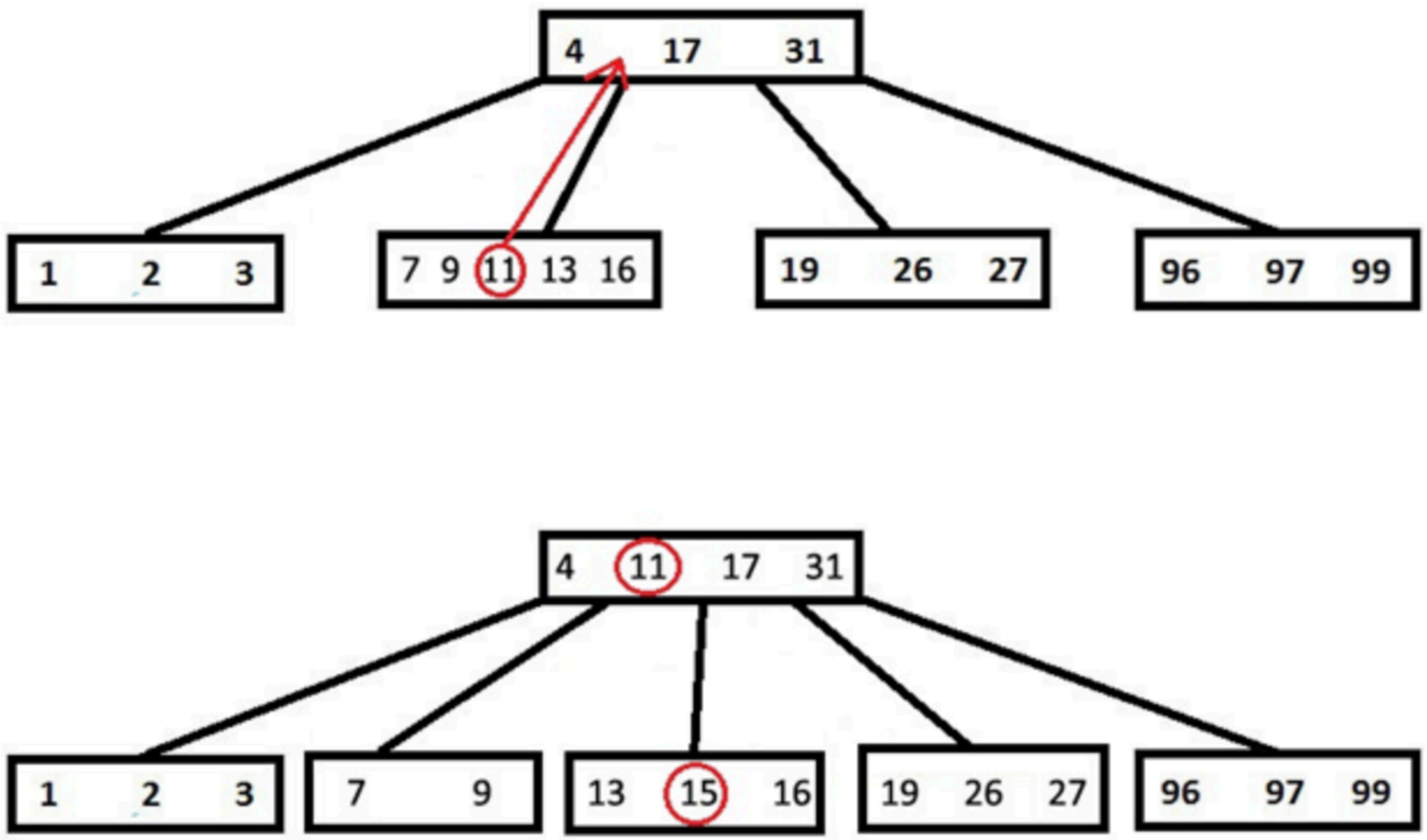
- **FOREIGN KEY**
- **Transactions**
- **ACID**
  - **Atomicity**
  - **Consistency**
  - **Isolation**
  - **Durability**
- **Buffer pool**
- **Change buffer**
- **Double write buffer**

- **Primary Key**
- **Unique**
- **Index**

B-tree Index

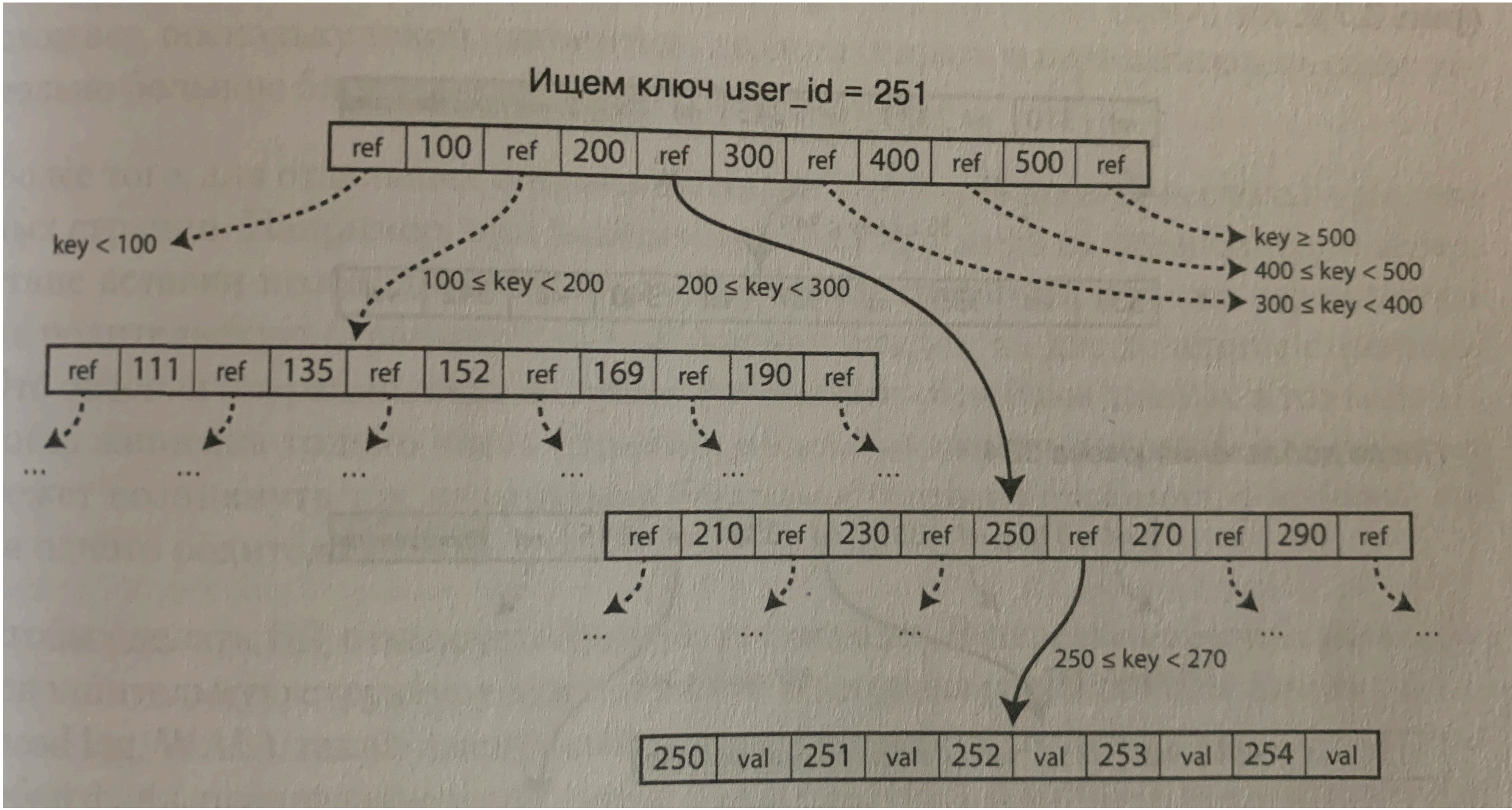


B-tree Index



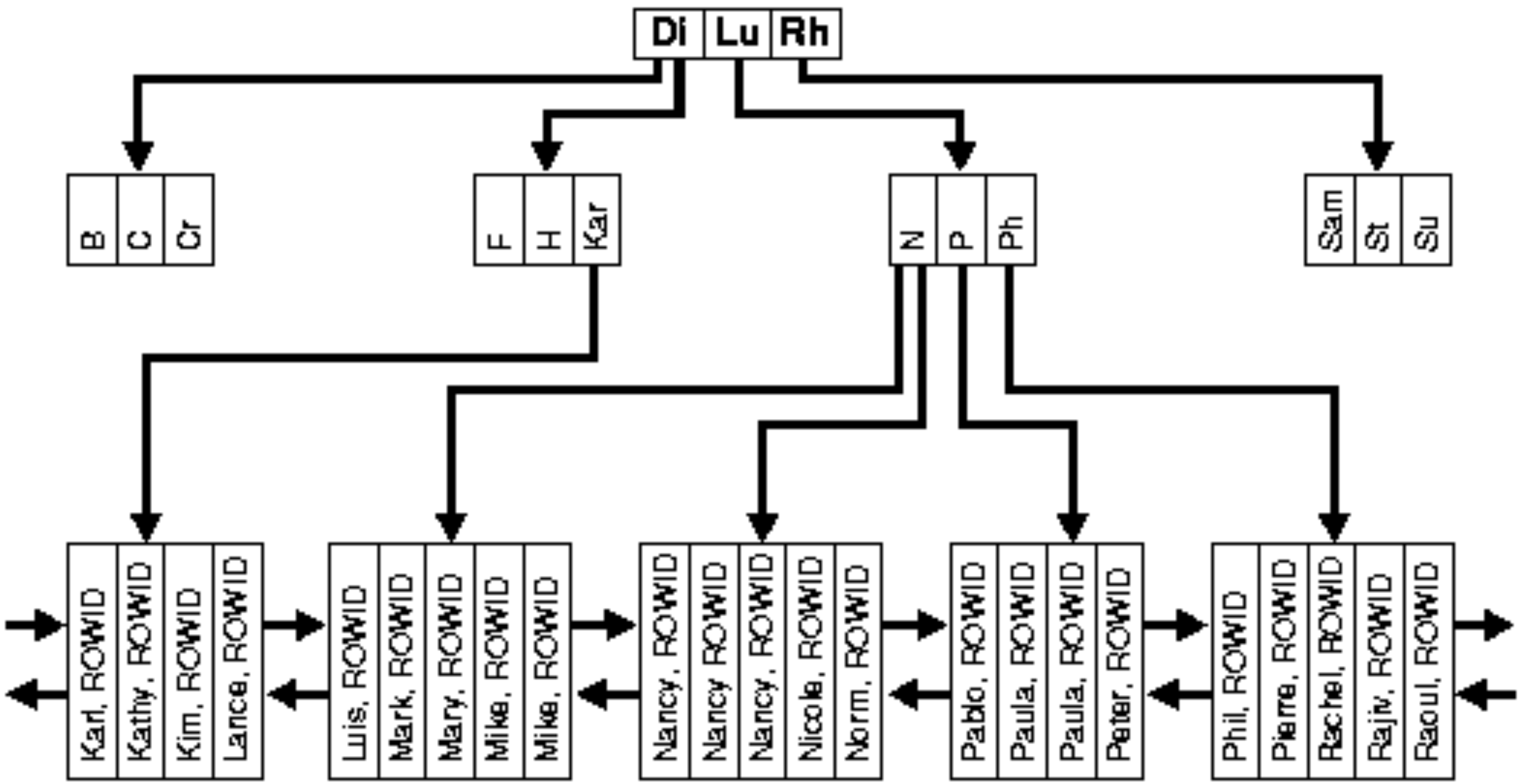
4 17 31
1 2 3
7 9 11 13 16
19 26 27
96 97 99
4 11 17 31
1 2 3
7 9
13 15 16
19 26 27
96 97 99

B-tree Index



Ищем ключ 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

B-tree Index



Di
Lu
Rh
B
C
Cr
F
H
Kar
N
P
Ph
Sam
St
Su
Karl, ROWID
Kathy, ROWID
Kim, ROWID
Lance, ROWID
Luis, ROWID
Mark, ROWID
Mary, ROWID
Mike, ROWID
Mike, ROWID
Nancy, ROWID
Nancy, ROWID
Nancy, ROWID
Nicole, ROWID
Norm, ROWID
Pablo, ROWID
Paula, ROWID
Paula, ROWID
Peter, ROWID
Phil, ROWID
Pierre, ROWID
Rachel, ROWID
Rajiv, ROWID
Raoul, ROWID

Complex index

```
SELECT * FROM users WHERE age = 29 AND gender = 'male'
```

```
CREATE INDEX age_gender ON users(age, gender);
```

```
SELECT * FROM users WHERE age => 10 AND gender = 'male'
```

```
CREATE INDEX age_gender ON users(gender, age);
```

Selectivity

The selectivity basically is a measure of how much variety there is in the values of a given table column in relation to the total number of rows in a given table

InnoDB

- **FOREIGN KEY**
- **Transactions**
- **ACID**
  - **Atomicity**
  - **Consistency**
  - **Isolation**
  - **Durability**
- **Buffer pool**
- **Change buffer**
- **Double write buffer**

Isolation levels

Initial state

```sql
CREATE TABLE accounts (
  id INT PRIMARY KEY,
  name VARCHAR(50),
  balance INT
);
INSERT INTO accounts (id, name, balance)
    VALUE (1, 'Alice', 100);
```

Isolation levels

READ UNCOMMITTED

A transaction can see changes to data made by other transactions that are not committed yet. This can lead to **dirty reads**.

Transaction A

```sql
-- Transaction A (any isolation level)
BEGIN;
UPDATE accounts SET balance = 200 WHERE id = 1; -- No commit yet
-- Some other operations
COMMIT;
```

Transaction B

```sql
-- Transaction B
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;

-- First Read (before Transaction A starts)
SELECT balance FROM accounts WHERE id = 1; -- Alice's balance
-- Output: 100

-- Second Read (Transaction A has updated but not committed)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 200 (reads uncommitted update)

-- Third Read (Transaction A has committed)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 200
```

Isolation levels

READ COMMITED

Only committed changes are visible. Prevents **dirty reads**.

Transaction A

```sql
-- Transaction A (any isolation level)
BEGIN;
UPDATE accounts SET balance = 200 WHERE id = 1; -- No commit yet
-- Some other operations
COMMIT;
```

Transaction B

```sql
-- Transaction B
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;

-- First Read (before Transaction A starts)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100

-- Second Read (Transaction A has updated but not committed)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100 (does not see uncommitted update)

-- Third Read (Transaction A has committed)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 200 (reads committed update)
```

Isolation levels

REPEATABLE READ (DEFAULT VALUE)

Each transaction has a consistent snapshot of data, so repeated reads yield the same result. Prevents dirty and non-repeatable reads.

Transaction A

```sql
-- Transaction A (any isolation level)
BEGIN;
UPDATE accounts SET balance = 200 WHERE id = 1; -- No commit yet
-- Some other operations
COMMIT;
```

Transaction B

```sql
-- Transaction B
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;

-- First Read (before Transaction A starts)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100

-- Second Read (Transaction A has updated but not committed)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100 (sees consistent snapshot, ignoring uncommitted changes)

-- Third Read (Transaction A has committed)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100 (still sees original snapshot)
```

Isolation levels

SERIALIZABLE

The strictest level, each transaction is fully isolated, as if transactions run one at a time.

Transaction A

```sql
-- Transaction A (any isolation level)
BEGIN;
UPDATE accounts SET balance = 200 WHERE id = 1; -- No commit yet
-- Some other operations
COMMIT;
```

Transaction B

```sql
-- Transaction B
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;

-- First Read (before Transaction A starts)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100

-- Second Read (Transaction A is trying to update)
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100 (Transaction A is blocked)

-- Third Read (Transaction A is still blocked until Transaction B finish
SELECT balance FROM accounts WHERE id = 1;
-- Output: 100
```

What we can use

- **Terminal**
- **DataGrip**
- **HeidiSQL**
- **Mysql Workbench**
- **DbViewer**

What to read

- 7 databases in 7 weeks
- High Performance MySQL, 4th Edition
- Designing Data-Intensive Applications - Kleppmann

Questions?