

DSA5104 Principles of Data Management and Retrieval

AY2025/26 Sem1 By Zhao Peiduo

Keys & Schemas

- Relation schema $R(A_1, \dots, A_n)$: instance $r(R)$ a table of tuples.
 - Superkey uniquely identifies tuples; candidate key = minimal superkey; Primary key chosen from candidate keys; foreign key references another relation.
- ```
CREATE TABLE dept(dept_name varchar(20) PRIMARY KEY);
CREATE TABLE instructor(ID varchar(5) PRIMARY KEY, dept_name varchar(20),
 FOREIGN KEY(dept_name) REFERENCES dept(dept_name));
```

## Relational Algebra → SQL

- $\sigma_p(r) \rightarrow \text{SELECT * FROM } r \text{ WHERE } p;$
- $\Pi_A(r) \rightarrow \text{SELECT DISTINCT A FROM } r;$
- $r \bowtie_{\rho} X = s \cdot X \rightarrow \text{SELECT * FROM } r \text{ JOIN } s \text{ ON } r.X=s.X;$
- $r \cup s, r \cap s, r - s \rightarrow \text{UNION / INTERSECT / EXCEPT}$
- $\rho_X(r) \rightarrow \text{FROM } r \text{ AS } x$

## SQL Database and Table Operations

- DDL/DML: schemas, domains, constraints; INSERT/DELETE/UPDATE, queries.
- Domains: char(n), varchar(n), int/smallint, numeric(p,d), double precision, float(n).
- Build-in Data Types: date('2025-7-27'), time('09:00:30.00'), timestamp('2025-7-27 09:00:30.00'), interval('1' day/year/hour/...), blob, clob.
- **INSERT:** insert into instructor values ('10211', 'Smith', 'Biology', 66000);
- User-defined types/domains with optional constraints: create type Dollars as numeric (12,2) final; create domain degree\_level varchar(10) check (value in ('Bachelors', 'Masters', 'Doctorate'));
- **Deferrable Foreign Key:** Postgres foreign key ... deferrable initially immediate/deferred; MySQL: set foreign\_key\_checks=0, insert, then set foreign\_key\_checks=1.
- **DELETE:** delete from student; (removes all tuples)
- MySQL Session Variable for DELETE: set @a = (select avg(salary) from instructor); delete from instructor where salary < @a;
- **DROP:** drop table r; drop database university;
- **ALTER:** alter table r add A D(domain); alter table r drop A;
- **Conditional UPDATE:** SET A = CASE ... END
- **Integrity Constraints:** primary key ( $A_1, \dots$ ); foreign key ( $A_m, \dots$ ) references r on (delete/update) (set null/default/cascade) not null; unique; check(predicate); To modify: alter table t add/drop constraint c

## SQL Query Keywords

- **Basic Query Ops:** SELECT, FROM, DISTINCT removes duplicates; ORDER BY sorts, BETWEEN v1 AND v2 for range filtering;
- **String Ops:** LIKE '%pat%' for pattern match. % for substring, \_ for single character, concat|| for combining strings, upper, lower for capitalization, length for string length, substr(str, start, end) for segmentation.
- **Joins:** FROM r, s WHERE r.X=s.X or explicit JOIN.
- **Set Ops:** UNION / INTERSECT / EXCEPT remove duplicates; UNION ALL keeps duplicates.
- **Aggregates:** avg, min, max, sum, count.
- **Grouping:** GROUP BY forms groups; HAVING filters groups; WHERE filters tuples before grouping; USING(A) specifies matched attributes; on <predicate> gives explicit join.
- **Set Membership:** IN/NOT IN, EXISTS/NOT EXISTS; SOME/ALL; UNIQUE.
- **Subquery Forms:** Subquery in FROM = temporary table; WITH defines CTEs: WHERE salary > ALL(SELECT salary FROM instructor WHERE dept.name='Biology');
- **NULL Test:** IS NULL / IS NOT NULL for tests. Comparisons with null yields unknown; WHERE treats unknown as false.

## Join Types

- **Self Join:** join a table with itself using aliases.
- **Natural Join:** matches on all common attributes; keeps one copy. Risky if unrelated attributes share names.
- **Inner Join:** only matching tuples.
- **Left/Right Outer:** keep all left/right tuples (other side padded with null).
- **Full Outer:** keep all tuples from both sides (MySQL: emulate with UNION).

## Views

- Virtual relation defined by query; stores expression, not data.
- Hides data and simplifies queries.  
    create view faculty as select ID, name, dept\_name from instructor;
- A view can be created from another view. Expanded at evaluation stage.
- Update possible only if view is based on one table, no aggregates/distinct.
- with check option ensures inserted/updated rows satisfy view condition.

**Transactions:** Atomic unit of work. MySQL autocommit on which can be disabled with texttstart transaction. End a transaction with commit or rollback.

**Indexes:** Improve lookup performance. MySQL auto-indexes PK + FK; PostgreSQL does not auto-index FK.

    create index studentID\_index on student(ID)

**B+ -Trees (Index):** Balanced; root-to-leaf same length. Leaves hold sorted search-key values. Each internal root has  $\lceil \text{ceil}(\frac{n-1}{2}), n-1 \rceil$  children. Each leaf node has  $\lceil \text{ceil}(\frac{n-1}{2}), n-1 \rceil$  values. Search the key by looking at internal nodes which are always sorted.

**Static Hashing:** Hash search keys and assign them to buckets; overflow handled with overflow buckets.

## Authorization

- Data privileges: select, insert, update, delete, all privileges.
- grant <privilege list> on <relation/view> to <User list>.
- revoke <privilege list> on <relation/view> from <User list>.
- Roles: distinguish users based on access or update, create role <name>.
- Other features: grant reference(dept\_name) on department to user; transfer of privileges: grant select on department to Amit with grant option; revoke select on department from Amit, Satoshi cascade/restrict.

**SQL/PSM:** Standard queries are declarative. SQL/PSM adds: BEGIN...END, variable DECLARE, IF, CASE, loops, exception handlers.

## Functions, Procedures, Triggers

- Function returns value, used in expressions.
- Procedure performs actions. Keywords in and out define in/out datatypes.
- A trigger is executed automatically as a side effect of a modification to the database.

## SQL in Python

```
import pymysql
connection_params = {'host': '127.0.0.1', 'user': 'root', 'password': 'ZQSLzwzw100',
 'database': 'university'}
with pymysql.connect(**connection_params) as conn:
 with conn.cursor() as cur:
 cur.execute('select * from instructor where salary > 90000')
 results = cur.fetchall()
 print(results)
```

## SQL Function Example

```
create function dept_count(dept_name varchar(20))
returns integer
begin
 declare d_count integer;
 select count(*) into d_count from instructor where instructor.dept_name = dept_name;
 return d_count;
end;
Usage: select dept_name, budget from department where dept_count(dept_name) > 12;
```

## Table Functions

```
create function instructor_of(dept_name varchar(20))
returns table(ID varchar(5), name varchar(20), dept_name varchar(20), salary numeric(8,2))
return table(select ID, name, dept_name, salary from instructor);
Usage: select * from table(instructor_of('Music'));
```

## MySQL Procedure

Delimiter in MySQL: Needed to define multi-statement procedures.

```
delimiter //
create procedure dept_count_proc(in dept_name_str varchar(20), out d_count int)
begin
 <omitted>
end //
delimiter ;
call dept_count_proc('Physics', @num_count);
select @num_count;
```

**Loops in SQL/PSM:** Use while, repeat, or labeled loop.

```
while boolean_expr do
 statements;
end while;
repeat
 statements;
until boolean_expr
end repeat;
```

## Language Constructs: if-then-else

```
if boolean_expression
 then statement
elseif boolean_expression
 then statement
else
 statement
end if;
```

## Handle Exception Condition

```
declare out_of_classroom_seats condition;
declare exit handler for out_of_classroom_seats
begin
 -- Exception handling logic
end;
signal out_of_classroom_seats; -- Raise exception
```

## Trigger in MySQL

```
delimiter //
create trigger trigger_name
 trigger_time(BEFORE/AFTER) trigger_event(INSERT/UPDATE/DELETE)
 on table_name for each row
begin
 ...
end //
delimiter ;
```

## Recursive Queries:

Handle recursive relationships. Use with recursive clause.  
with recursive rec\_prereq(course\_id, prereq\_id) as (
 select course\_id, prereq\_id from prereq
 union
 select p.course\_id, r.prereq\_id from prereq p, rec\_prereq r
 where p.prereq\_id = r.course\_id
)

select \* from rec\_prereq;

## Window Functions

- Aggregate: SUM, COUNT, AVG, VARIANCE, STDDEV, MIN, MAX.
- Non-aggregate: RANK, DENSE\_RANK, ROW\_NUMBER, NTILE(n), PERCENT\_RANK, CUME\_DIST, ROW\_NUMBER.

```
window_function(expr) over ([partition by ...] [order by ...]
 [frame clause: rows <n>/unbounded preceding/following/current row,
 BETWEEN clause_1 AND clause_2])
```

## Cross Tabulation (Pivot Table)

```
select item_name,
 sum(case color when 'dark' then quantity end) as dark,
 sum(case color when 'pastel' then quantity end) as pastel,
 sum(case color when 'white' then quantity end) as white
from sales
group by item_name;
```

## Rollup in MySQL

```
select item_name, color, sum(quantity)
from sales
group by item_name, color with rollup;
```

**Application Programs and Web:** The architecture contains frontend(UI), middle layer (logic, security) and backend(DB access)

## URL Components:

scheme://domain:port/path?parameters#anchor

## World Wide Web

HTML pages contain text, links, and forms. Example:

```
<!DOCTYPE html>
<head> <title> Sample form </title> <meta charset="utf-8"> </head>
<html><body> <!--Displayed content--><form action="PersonQuery" method="get">
 <select name="persontype">
 <option value="student" selected>Student</option>
 <option value="instructor">Instructor</option>
 </select>

 Name:<input type="text" size=20 name="name">
 <input type="submit" value="submit">
</form> </body> </html>
```

## Client-side Scripting: JavaScript Validation Example

```
<script>
function validate() {
 var c=document.getElementById("credits").value;
 if (isNaN(c) || c<=0 || c>=16) { alert("Invalid"); return false; }
}
</script>
<form action="createCourse" onsubmit="return validate()">
```

**Functional Dependencies:**  $X \rightarrow Y$ : same  $X$  implies same  $Y$ . It is not symmetric, based on semantics and applies to all instances.  $F^+$ : all FDs implied by  $F$ .

## Rules of Inference

- Reflexivity: If  $X \subseteq Y$ , then  $X \rightarrow Y$
- Augmentation: If  $X \rightarrow Y$ , then  $XZ \rightarrow YZ$
- Transitivity: If  $X \rightarrow Y$  and  $Y \rightarrow Z$ , then  $X \rightarrow Z$
- Union:  $X \rightarrow Y$  and  $X \rightarrow Z \Rightarrow X \rightarrow YZ$
- Decomposition:  $X \rightarrow YZ \Rightarrow X \rightarrow Y$  and  $X \rightarrow Z$
- Pseudotransitivity:  $X \rightarrow Y$  and  $QY \rightarrow Z \Rightarrow QX \rightarrow Z$

## Computing $F^+$

```
F+ = F
apply reflexivity
repeat
 for each FD f in F+: apply augmentation
 for each pair f1,f2 in F+: apply transitivity
 until F+ does not change further
```

**Attribute Closure:**  $X^+$ : all attributes functionally determined by  $X$ . Check if  $X \rightarrow Y \in F^+$  by testing if  $Y \subseteq X^+$ . Attribute closure algorithm:

```
X* := X
repeat
 for each U \rightarrow V in F
 if U \subseteq V then X* := X* \cup V
 until X* does not change
```

## Use of Attribute Closure

- Superkey test: If  $X^+$  contains all attributes then  $X$  is a superkey.
- Candidate key test:  $X$  is a superkey and for every attribute  $A \in X$ ,  $(X - A)^+ \neq R \Rightarrow X$  is a candidate key.

**Functional dependency test:** If  $Y \subseteq X^+$ , then  $X \rightarrow Y$  holds.

- Computing the closure of  $F(F^+)$ : For each  $X \subseteq R$ , compute  $X^+$ . For each  $S \subseteq X^+$ , output the functional dependency  $X \rightarrow S$ .

## Normal Forms

- 1NF: atomic (no composite values/relations) + no duplicate tuples/records.
- Boyce-Codd(BC)NF: for every nontrivial  $X \rightarrow Y$ ,  $X$  must be a superkey. (or, either  $A \subseteq X$  (trivial) or  $X$  is a superkey).

**Decomposition:** Replace  $R$  with schemas containing attribute subsets. Must preserve all attributes. Lossless if replacing  $R$  with  $X \cup Y$  preserves all information i.e.  $\pi_X(r) \bowtie \pi_Y(r) = r$

**Lossless Join Condition:** Ensures reconstructing  $R$  via joins does not introduce spurious tuples.

•  $(X \cap Y) \rightarrow X$  or  $(X \cap Y) \rightarrow Y$

• Corollary: If  $X \rightarrow Z$  and  $X \cap Z = \emptyset$ , then  $R - Z$  and  $XZ$  is lossless.

Testing for lossless join property: Initialize the table with  $b$  values, cancel all LHS variables in all FDs and update all  $b$  values that appear on RHS suppose there is a row with both LHS and RHS as  $a$  values.

- Create matrix  $S$  with rows for relations  $R_i$  and columns for attributes  $A_j$ .
- Initialize entries  $S(i,j)$  with distinct symbols.
- For each  $R_i$  containing  $A_j$ , set  $S(i,j) = a_j$ .
- Repeat until no change: For each FD  $X \rightarrow Y$ , enforce same symbols in  $X$  columns imply same in  $Y$ .

• If any row becomes all  $a_j$ , decomposition has nonadditive join property.

## Dependency Preservation Decomposition

- Projection  $F_X$ : FDs in  $F^+$  using only attributes in  $X$ .

- Decomposition is dependency preserving if  $(F_X \cup F_Y)^+ = F^+$ .

## Testing for Dependency Preservation

- Compute  $F^+$ .
- For each schema  $R_i$  in  $D$ , let  $F_i$  = projection of  $F^+$  onto  $R_i$ .
- Let  $F' := \bigcup_i F_i$ .
- Compute  $F'^+$ .
- If  $F'^+ = F^+$  then decomposition is dependency preserving, else not.

**Alternative One (Sufficient condition):** If each FD in  $F$  can be checked on one relation in the decomposition, then dependency preserving.

**Alternative Two** Apply the following to all  $a \rightarrow b$ :

```
result = a
repeat
 for each Ri in the decomposition
 t = (result INTERSECT Ri)* INTERSECT Ri
 result = result UNION t
 until (result does not change), result should contain all attributes in b.
Tips: Filter FDs with alternative 1, and pick a violating FD, run alternative 2 to show it violates dependency preservation.
```

**BCNF Decomposition Algorithm:** Ensures lossless join, but may lose dependency preservation.

1. Find FD  $X \rightarrow Y$  that violates BCNF.
2. Decompose  $R$  into  $X+$  and  $X \cup (R - X+)$ .
3. Repeat until all schemas in BCNF.

#### Semi-Structured Data Types

- Composite(has-sub-attributes): {first, last}
- Multivalued (list/set): hobbies = ["piano", "basketball"]
- Maps: {brand: Apple, size:13}

#### RDF (Resource Description Framework)

- Data as triples: (subject, predicate, object). Attribute facts(ID, attr, value), and Relationship facts: (ID1, rel, ID2), where ID is identifiers of entities.
- Naturally forms a graph: Objects = ovals, attribute values = rectangles, relationships = edges with associated labels identifying the relationship.

#### SPARQL Example: Find students' names taking "Intro to CS"

```
SELECT ?name WHERE {
?cid :title "Intro to CS". # course ID with the title
?sid :sec_course ?cid. # section belongs to that course
?id :takes ?sid. # student takes that section
?id :name ?name .
}
```

#### XML (eXtensible Markup Language) Basics

- Elements: <tag> ... </tag>, properly nested with subelements.
- Attributes: metadata and identifiers. e.g. <course id="CS-101">. name = "value" where value must be quoted, names must be unique and appear at most once. All data represented as text.
- One root element per document. Mixed content is legal but discouraged.
- Empty element: <item/> (start and end tags combined together).

#### XML Namespaces

- Avoid naming conflicts. xmlns:prefix="URI" binds prefixes. Default: xmlns="URI"
- Meaning determined by URI, not prefix text.

```
<university xmlns="http://example.org/univ" xmlns:yale="http://www.yale.edu/ns">
<course>
 <title>Intro to Computer Science</title>
 <code dept="CS" number="101"/>
 <code:course>
 <yale:title>Intro to CS</yale:title>
 <yale:code yale:dept="CS" yale:number="101"/>
 </yale:course>
</course>
</university>
```

**XML CDATA:** To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA: Starts with <![CDATA[ and ends with ]>. For example: <![CDATA[<course> ... </course>]]>.

#### XML Schema - DTD (Document Type Definition)

- Constrains structure: Restrict on the presence of elements and attributes and frequencies. ID: unique within document. IDREF(S): reference existing IDs.
- No data typing; all text. Order of subelements is significant and need to list all permutations. Syntax:

```
<ELEMENT element (subelements-specification: name of elements, #PCDATA, EMPTY, ANY)>
Example: <ELEMENT dept (dept_name, building, budget)>, can be extended by regex (,|*?)>
<!ATTLIST element (attribute: Name, Types(CDATA/ID/IDREF),
Defaults(#REQUIRED/#IMPLIED/"default_value")>
Example: <!ATTLIST dept dept_name CDATA #REQUIRED>
```

#### XML Schema (XSD)

```
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema" targetNamespace="http://ex.org/u
xmsns="http://ex.org/u" elementFormDefault="qualified"><!-- namespace declaration -->
<xss:simpleType name="PosInt"><!-- simple type restriction -->
 <xss:minInclusive value="0"/>
</xss:restriction>
</xss:simpleType>
<xss:attributeGroup name="CommonMeta"><!-- attribute group reused -->
 <!-- use: optional|required/prohibited -->
 <xss:attribute name="uuid" type="xs:ID" use="required"/>
 <xss:attribute name="active" type="xs:boolean" default="true"/>
</xss:attributeGroup>
<xss:element name="DeptCode" type="xs:string"/>
<xss:complexType name="DeptType"><!-- sequence + choice + attributes -->
 <xss:sequence>
 <xss:element name="dept_name" type="xs:string"/>
 <xss:element name="building" type="xs:string" minOccurs="0"/> <!-- occurrence -->
 <xss:element ref="u:DeptCode" minOccurs="1"/><!-- global reference -->
 </xss:sequence>
 <xss:attribute name="code" type="xs:string" use="required"/>
 <xss:attributeGroup ref="u:CommonMeta"/>
</xss:complexType>
<!-- Instructor hierarchy (substitution group example) -->
<xss:element name="person" abstract="true"/>
<xss:element name="instructor" substitutionGroup="u:person">
<xss:complexType>
 <xss:sequence>
 <xss:element name="name" type="xs:string"/>
 <xss:element ref="u:DeptCode"/> <!-- consistent reference -->
 </xss:sequence>
<xss:attributeGroup ref="u:CommonMeta"/>
</xss:element>
<xss:complexType name="CourseType"><!-- restricted type + referencing -->
 <xss:sequence>
 <xss:element name="title" type="xs:string"/>
 <xss:element name="credits" type="u:PosInt"/>
 <xss:element ref="u:DeptCode"/> <!-- reusing deptCode, below nullable -->
 <xss:element name="comment" type="xs:string" nullable="true" minOccurs="0"/>
 </xss:sequence>
 <xss:attribute name="id" type="xs:ID" use="required"/>
</xss:complexType>
```

```
<xss:element name="university"> <!-- root container -->
<xss:complexType>
 <xss:sequence>
 <!-- multiple departments, courses, instructors via substitution group -->
 <xss:element name="department" type="u:DeptType" maxOccurs="unbounded"/>
 <xss:element name="course" type="u:CourseType" maxOccurs="unbounded"/>
 <xss:element ref="u:person" maxOccurs="unbounded"/>
 </xss:sequence>
</xss:complexType>
<xss:key name="deptKey"> <!-- key: dept_code must be unique -->
<xss:selector xpath="u:department"/>
<xss:field xpath="u:deptCode"/>
</xss:key>
<!-- keyref: course.deptCode must reference key -->
<xss:keyref name="courseDeptRef" refer="deptKey">
<xss:selector xpath="u:course"/>
<xss:field xpath="u:deptCode"/>
</xss:keyref>
<!-- keyref: instructor.deptCode must reference same key -->
<xss:keyref name="instructorDeptRef" refer="deptKey">
<xss:selector xpath="u:instructor"/>
<xss:field xpath="u:deptCode"/>
</xss:keyref>
</xss:element>
</xss:schema>
```

**XML Tree Model:** Elements = nodes; attributes = node properties. Each node has 1 parent; children are ordered.  
**XPath:** Select nodes via paths. // all descendant, .. all parents, @ attribute, \* wildcard, | union expressions results, doc(name) root of a named document.

```
/uni/course[@course_id] /uni/course/credits /uni/course[credits > 4]/@course_id
/uni/course/position()<3> (count(), position(), not(), boolean and/or)
/uni/course/id(@dept_name) (id) resolves ID/IDREFs
```

**XQuery FLWOR:** FOR,LET,WHERE,ORDER BY,RETURN. Curly braces evaluate expressions inside constructed XML. Sequence Comparisons: General: '=' , '<' , '>' (true if any pair matches). Value: 'eq', 'lt', ... (require single value).<br/><!-- Join -->

```
for $c in /uni/course, $i in /uni/instructor, $t in /uni/teaches
```

```
where $c/id = $t/cid and $t/iid = $i/id
```

```
return <course_instructor>$c $i</course_instructor>
```

<!-- Function and invocation -->

```
declare function local:dept_courses($id as xs:string) as element(course)* {
```

```
 for $i in /university/instructor[IID = $id],
```

```
 $c in /university/course[dept_name = $i/dept_name]
```

```
 return $c
```

```
};
```

```
for $i in /university/instructor[name = "Sophie"]
```

```
return local:dept_courses($i/IID)
```

<!-- Grouping / Aggregation -->

```
for $d in /university/department
```

```
return
```

```
<dept_total_salary>
```

```
<dept>{ $d/dept_name}</dept>
```

```
<total> fn:sum(for $i in /university/instructor[dept_name=$d/dept_name]
 return $i/salary)
```

</total>

```
</dept_total_salary>
```

<!-- Quantifiers: some, every, fn:exists(xpath) -->

```
for $d in /university/department
```

```
where every $i in /university/instructor[dept_name=$d/dept_name] satisfies $i/salary > 5000
```

```
return $d
```

<!-- Control Flow -->

```
for $p in /products/product[name="Gadget"]
```

```
return
```

```
 if ($p/price > 100) then
```

```
 <result>The product is expensive</result>
```

```
 else
```

```
 <result>The product is affordable</result>
```

SQL/XML extension

<!-- xmlelement, xmllistattributes -->

```
select xmlelement(name="course", xmllistattributes(course_id, dept_name),
```

```
xmlelement(name="title", title), xmlelement(name="credits", credits)) from course;
```

<!-- xmllag: XML GROUP BY -->

```
select dept_name, xmllag(xmloffset(course_id) order by course_id)
```

```
from course group by dept_name;
```

**Storing XML in Databases:** Flat file/String storage (CLOB): simple, but parsing slow for string. Tree storage (nodes table): flexible, but data is broken up into too many pieces requiring more joins, more space overheads. **Relational mapping:** tables per element type; FKs preserve hierarchy, but have overhead of translating data and queries.

**NoSQL Taxonomy:** Key-Value: Redis, DynamoDB (fast lookup). Document: MongoDB, CouchDB (JSON/BSON).Graph: Neo4j (relationships).Vector DBs: Pinecone (similarity search).

**CAP Theorem:** Cannot guarantee all: **Consistency**, **Availability**, **Partition Tolerance**. Cloud systems require P → choose CP or AP. CP = accuracy; AP = performance with eventual consistency.

**MongoDB Basics:** Document store, BSON, schema-flexible. Designed for horizontal scaling (more machines, sharding, as compared to vertical/bigger machine). Typically CP.

**Eventual Consistency:** Replicas may return stale reads temporarily. Eventually all converge if no new writes.

**MongoDB Data Model Database** → Collections → Documents(supports nested objects + arrays). id: primary key (unique, immutable, may be any non-array type. Default = ObjectId, ordered by time).

**JSON vs BSON** JSON: text-based, flexible schema, field-pairs and ordered lists. BSON: binary, supports Date, Decimal128, ObjectId, lightweight, traversable, efficient, optimized for performance and storage.

#### CRUD Summary

```
-- Basics --
use mydb
show collections
db.createCollection("movies")
-- Create, Update, Delete --
db.movies.insertOne({ title: "Inception", year:2010 })
db.movies.insertMany([{ title: "Dune", year:2021 }, { title: "Arrival", year:2016 }])
db.movies.updateOne({ year: { $lt: 2000 } }, { $set:{rating:8.2}, $currentDate:{updated:true} })
db.movies.replaceOne({ title: "Dune" }, { title: "Arrival", director: "Denis Villeneuve", rating:8.0 })
db.movies.deleteOne({ title: "Arrival" })
db.movies.deleteMany({ year: { $lt: 1980 } })
-- Read --
db.movies.find({ year: 2010 })
db.movies.find({ year: { $gte: 2010, $lte: 2020 } })
db.movies.find({ genres: { $in: ["Drama", "History"] } })
db.movies.find(
 { and: [
 { year: 1995 },
 { $or: [{ director: "Nolan" }, { title: "Inception" }] }
]
)
db.movies.find({ year:2018 }, { title:1, year:1, _id:0 }) -- Projection --
db.inventory.find({ item: { $type: "null" } }) -- null test-
db.inventory.find({ item: { $type: 10 } }) -- null test-
-- Array Queries --
db.inventory.find({ tags: { "$red": "blank" } }) -- in exact order --
db.inventory.find({ tags: { $all: ["red", "blank"] } }) -- any order --
db.inventory.find({ instock: { $qty: { $gte:20 } } })
db.inventory.find(
 { instock: { $elemMatch: { qty: { $gt:20 }, warehouse:"A" } } }
)
-- Single Purpose Aggregation Operations --
db.inventory.estimatedDocumentCount() / db.inventory.count()
db.inventory.distinct("item") -- find distinct items
```

#### Aggregation Pipeline Example

```
db.movies.aggregate([
 { $match: { year: 2020 } },
 { $group: { _id: "$genre", avgRating: { $avg: "$imdb.rating" } } },
 { $project: { genre: "$_id", avgRating: 1, _id: 0 } },
 { $sort: { avgRating: -1 } },
 { $limit: 5 }
])
```

#### Query Operators

```
Find: $and, $or, $gt(e), $lt(e), $ne, $in, $all, $size (of array), $elemMatch,
$slice (+n first n elements; -n: last n elements), $exists, $type
Update: $currentDate, $inc, $min, $max, $mul, $rename, $setOnInsert, $unset
Aggregation pipelines:
$set: Add/modify fields, $count,
$out/merge: overwrite/join a target collection (create if not exists), must be last stage.
Indexes: B-tree structure, O(log n) lookup .Create:
db.users.createIndex({ score: 1 })
db.users.createIndex({ user:id, score:-1 }) // compound
db.users.createIndex({ "tags":1 }) // multikey
db.users.dropIndex({ score:1 }) // force using ascending score index
```

#### Text Search

```
db.products.createIndex({ keywords:"text" })
db.products.find({ $text:{ $search:"laptop Apple" } })
```

**Schema Design Embedding** (store inside parent): fast reads, atomic updates. **Reference** (store ObjectId links): scalable, flexible. Rule: If "many" grows large → use references.

**Big Data:** Traditional DB systems cannot scale to massive, fast, heterogeneous data. 5 V's: Volume, Variety, Velocity, Veracity, Value.

**Hadoop Ecosystem:HDFS:** distributed file storage. **YARN:** cluster resource management. **MapReduce:** distributed processing engine.

**HDFS:** **NameNode:** master; stores metadata + block locations. **DataNodes:** store blocks; send heartbeats. Files split into blocks (default **128MB**) and replicated across nodes. **Rack-aware** placement → fault tolerance.

#### MapReduce Model:

- **map:** (K1, V1) → list of (K2, V2). **shuffle + sort:** group by key. **reduce:** (K2, list(V2)) → output.
- **Combiner:** Local mini-reducer to reduce shuffle volume. Must be associative + commutative.
- **Reduce-side join:** one table must fit in memory; avoids shuffle; **fast**.

**Spark Motivation:** MapReduce is slow due to repeated disk I/O. Works poorly for iterative / interactive workloads.

**Spark:** Resilient Distributed Dataset(RDD) = Resilient(track data lineage information to recover lost data automatically on failure), immutable(cannot be changed partially), lazy(transformations are actually computed when you call action) parallel and distributed dataset with lineage. **In-memory** processing → 10–100× faster than MapReduce. **Lazy evaluation** → optimizes execution DAG. Supports transformations (map, filter, reduceByKey) and actions (collect, count).

**Components of spark:** Spark SQL and Dataframe - structured data processing; Spark Streaming - real-time data processing; MLlib - scalable machine learning library; GraphX - graph data processing.

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.appName("Python Spark SQL basic example").getOrCreate()
df = spark.read.json("./sample_data/people.json")
df.show()
df.printSchema()
df.select("name").show()
df.select(df['name'], df['age']+1).show()
df.filter(df['age'] > 21).show()
```