SQL

Spring 2025

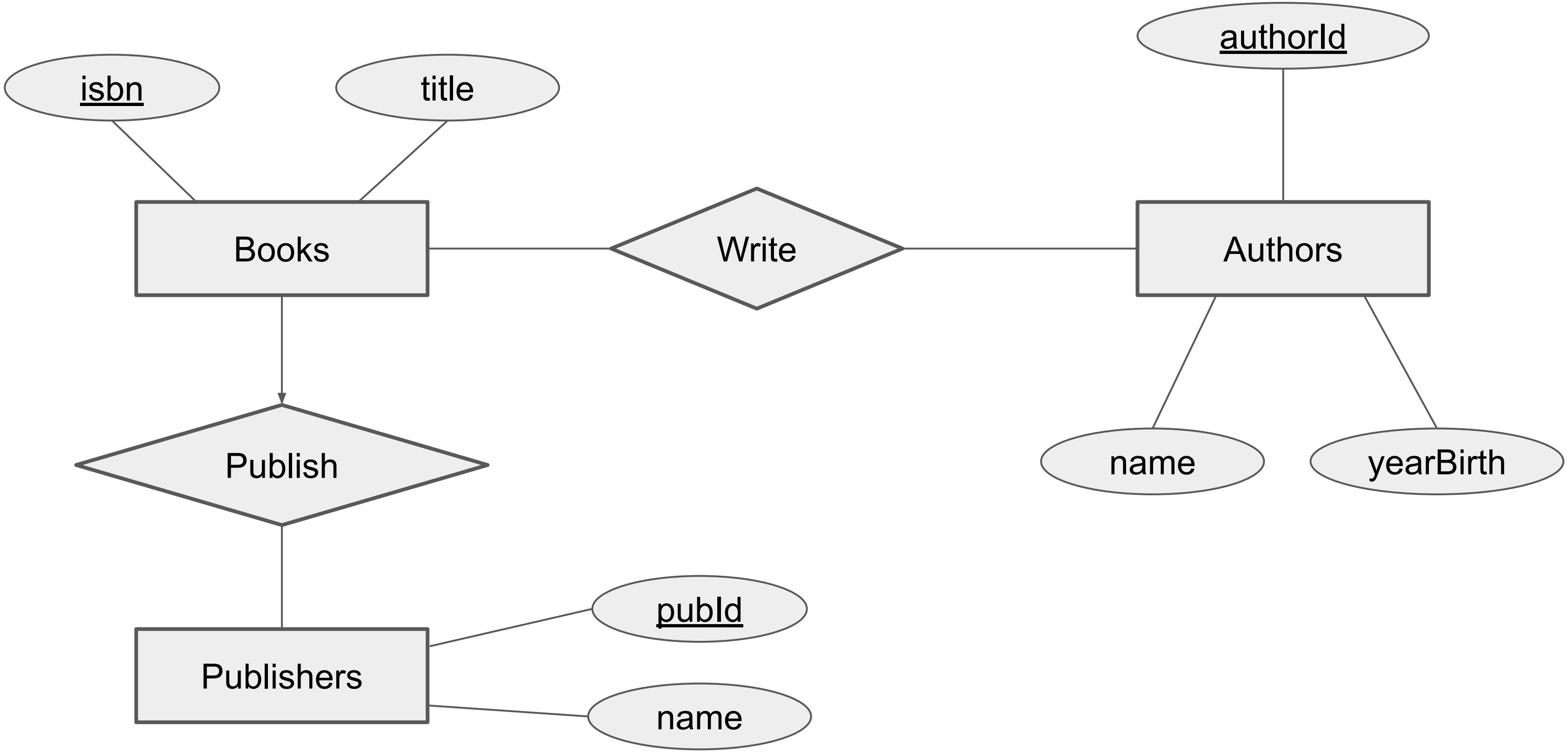
Last time

Conceptual model	What is the data about?
Logical model	**How do we represent the data in a specific (kind of) database?**
Physical model	How is the data represented in memory or on disk?

Last time

Relational databases:

- records in tables
- easy to understand
- flexible enough for most data uses
- supports a powerful query language (SQL)


isbn
title
authorId
Books
Write
Authors
Publish
name
yearBirth
pubId
Publishers
name

Books

foreign key

isbn	title	year	pubId
0771595565	Rebel Angels	1981	101
0316296198	The Magus	1965	102
0670312134	Fifth Business	1970	101

Write

isbn	authorId
0771595565	1
0316296198	2
0670312134	1

Authors

authorId	name	yearBirth
1	Robertson Davies	1913
2	John Fowles	1926

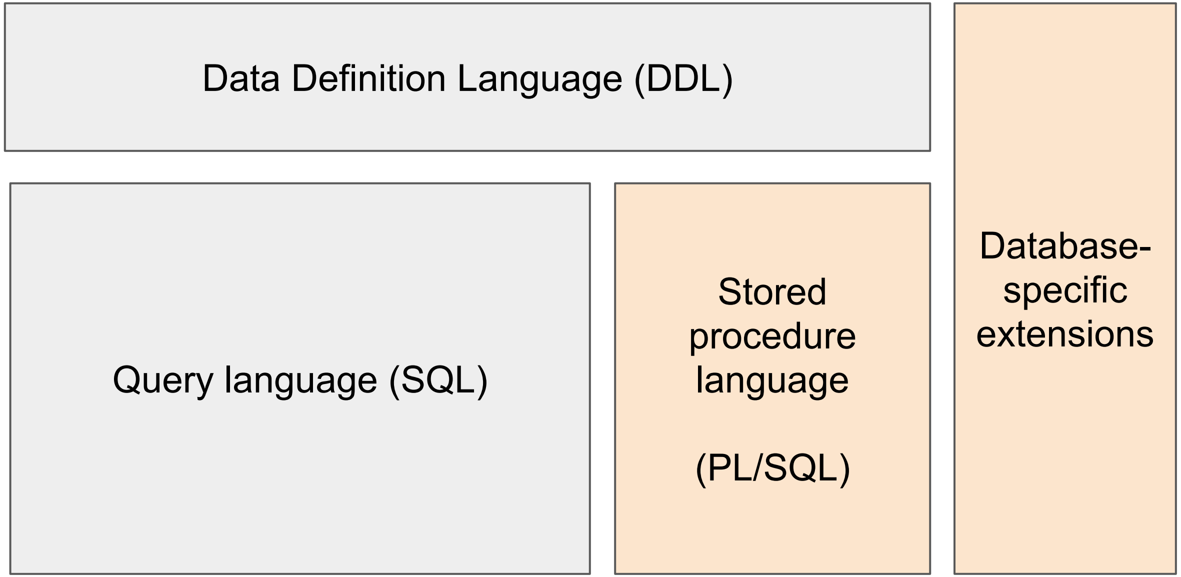
Publishers

pubId	name
101	McMillian
102	Little Brown & Co

~~isbn~~	~~pubId~~
~~0771595565~~	~~101~~
~~0316296198~~	~~102~~
~~0670312134~~	~~101~~

Can we also remove table Write, while retaining the ability to have multiple authors per book?

SQL at a glance


Data Definition Language (DDL)
Query language (SQL)
Stored procedure language

(PL/SQL)
Database-specific extensions

Data Definition Language

Subset of SQL for table CRUD operations

Create a table:

```
CREATE TABLE name (...)
```

Update a table (structurally — add column, etc)

```
ALTER TABLE name ...
```

Delete a table:

```
DROP TABLE name
```

Query language

Add a row to a table:

INSERT INTO *table* **VALUES** (*value*$_1$, *value*$_2$, …)

Update rows in a table

UPDATE *table* **SET** *column* = *value* **WHERE** *row condition*

Delete rows in a table

DELETE FROM *table* **WHERE** *row condition*

Read rows from a table

SELECT *column*$_1$, *column*$_2$, … **FROM** *table*
SELECT *column*$_1$, *column*$_2$, … **FROM** *table* **WHERE** *row condition*

Querying via SELECT

Querying = extracting information out of the data that's stored in the database

Key operation for relational databases, and where SQL puts most of its focus

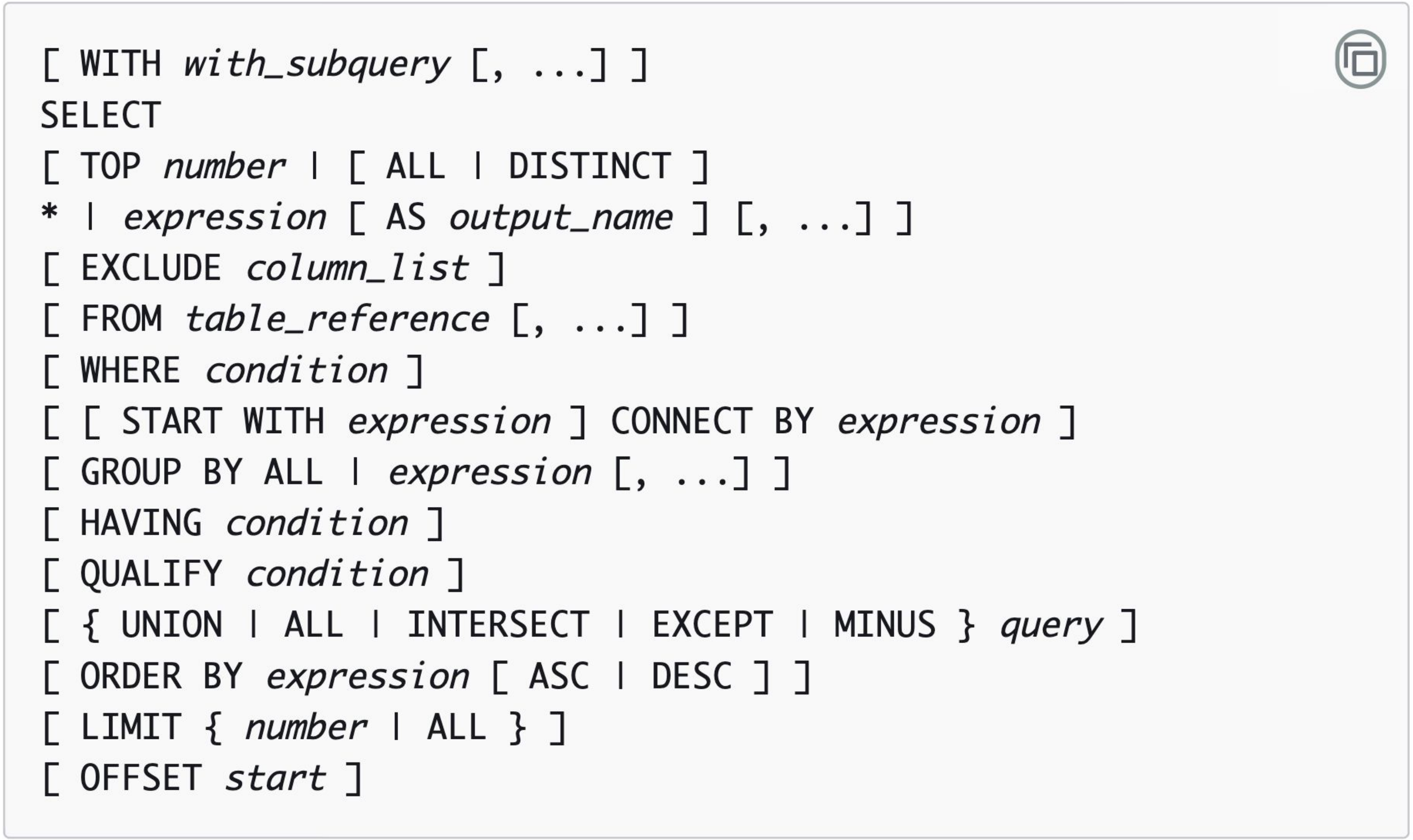
- store data on a per-table basis
- retrieve data across tables

Keep as much of the processing on the database

- you should filter / massage the data on the DB side as much as possible
- (imagine having to pull TBs of data just to access a single row)

Syntax

https://docs.aws.amazon.com/redshift/latest/dg/r_SELECT_synopsis.html



```
[ WITH with_subquery [, ...] ]
SELECT
[ TOP number | [ ALL | DISTINCT ]
* | expression [ AS output_name ] [, ...] ]
[ EXCLUDE column_list ]
[ FROM table_reference [, ...] ]
[ WHERE condition ]
[ [ START WITH expression ] CONNECT BY expression ]
[ GROUP BY ALL | expression [, ...] ]
[ HAVING condition ]
[ QUALIFY condition ]
[ { UNION | ALL | INTERSECT | EXCEPT | MINUS } query ]
[ ORDER BY expression [ ASC | DESC ] ]
[ LIMIT { number | ALL } ]
[ OFFSET start ]
```

Understanding queries

Four basic operations:

1. projecting
2. filtering
3. joining
4. aggregating

These operations form the basis of so-called **relational algebra** that formalizes the concept of queries in the context of relations

- relational algebra = mathematical basis of SQL queries

Intuition

A query **pulls rows from one or more tables** and **returns the rows of a new (virtual) table** representing the result of the query

- that result table is transitory and not persisted on disk
- a query just returns rows to the client issuing the query

That means that you can generally put a query wherever SQL expects a table

- nested queries

You can also create an actual table out of the results of the query

CREATE TABLE *table* **AS (** *query* **)**

Projecting

Return **narrower** rows from a source table

SELECT *column*$_1$, *column*$_2$, …
FROM *table*

Same number of rows as the source table

Special case (all columns):

SELECT *
FROM *table*

Rename columns in the result

SELECT *column*$_1$ **AS** *name*$_1$, … **FROM** *table*

Filtering

Happens before projecting

Remove some of the rows from the source table

SELECT *column*$_1$, … **FROM** *table* **WHERE** *row condition*

where *row condition* is a condition that is true or false of a row of the source table

There is a full language of conditions

- *column = value, column*$_1$ = *column*$_2$, *column > v*
- more generally: *exp*$_1$ *op exp*$_2$
- *column* **IS NULL**, *column* **IS NOT NULL**
- Boolean combinations: **AND**, **OR**, **NOT**

(Fun fact)

What if you wanted to project *before* filtering?

- usually doesn't make a difference, but for the sake of argument…

Use a nested query:

```
SELECT * FROM (
    SELECT column_1, column_2 FROM table
)
WHERE row condition
```

Joining

Until now, I've use a single source table for queries

We can also **pull rows from multiple tables** in a query

- what does that even mean?
- (conceptually) **join the tables into a single table** before querying from it
- happens before filtering and projecting

Join the tables?

- many possible interpretations

Joining — Cartesian joins

Simplest join — consider all possibilities

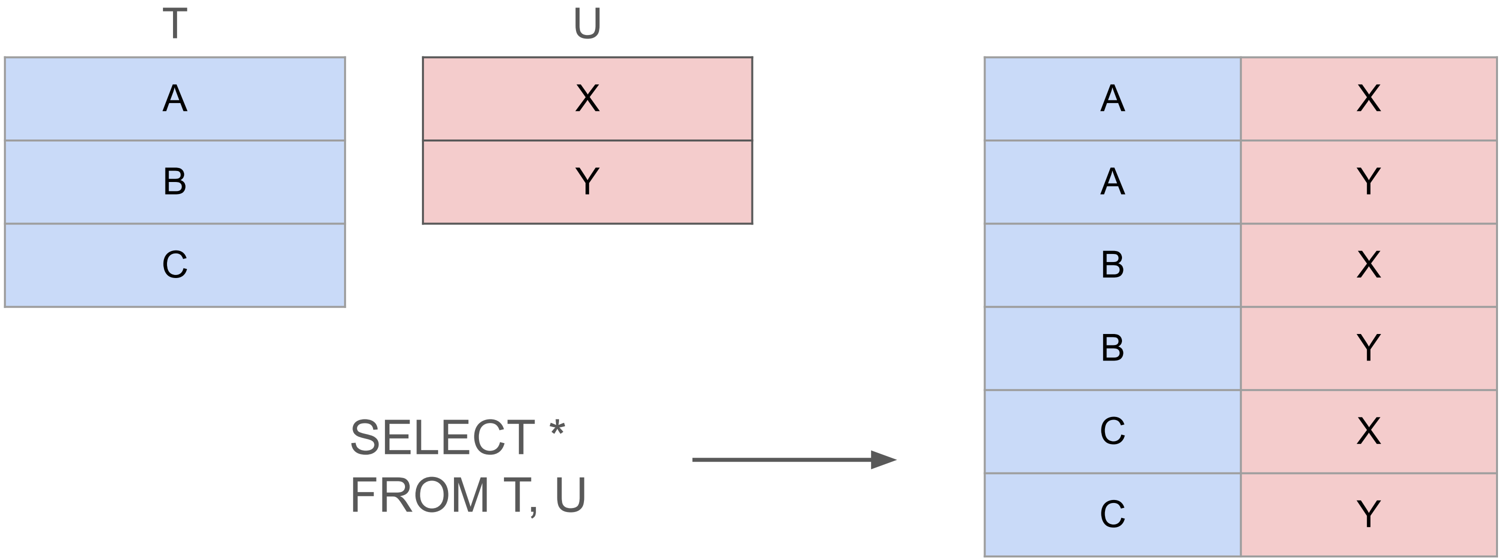
```
SELECT *
FROM Books, Authors
```

→ take every possible row from A and every possible row from B
→ concatenate each combination into a (longer) row
→ total number of rows in result = |tableA| |table B|

Generalizes to an arbitrary number of tables:

```
SELECT *
FROM Books, Write, Authors
```

Example (Cartesian join)



T
A
B
C

U
X
Y

```
SELECT *
FROM T, U
```

A	X
A	Y
B	X
B	Y
C	X
C	Y

Joining — inner joins

Almost always, you want to **match** the rows from the tables

- a row from table A has a foreign key into a row of table B
- *"instead of going from table A row to the table B row, attach B row to A row"*

You can get that with filtering:

```
SELECT Books.title, Authors.name
FROM Books, Authors, Write
WHERE Books.isbn = Write.isbn AND Authors.authorId = Write.authorId
```

Same row may be matched multiple times (book with multiple authors)

Joining — inner joins

Alternative syntax:

```
SELECT Books.title, Authors.name
FROM Books
[INNER] JOIN Write ON Books.isbn = Write.isbn
[INNER] JOIN Authors ON Authors.authorId = Write.authorId
```

Special case (**natural join**):

```
SELECT Books.title, Authors.name
FROM Books
[INNER] JOIN Write USING (isbn)
[INNER] JOIN Authors USING (authorId)
```

Example (inner join)

T

	j
A	1
B	2
C	

U

k	
1	X
1	Y
	Z

```
SELECT *
FROM T
JOIN U on T.j = U.k
```

→

A	1	1	X
A	1	1	Y

Joining — outer joins

if a **JOIN** field value does not match for a row, the row will not show up in the **JOIN**

- this includes null values (not equal to any other value)

What if you wanted unmatched rows to be included?

- what do you join it with? nothing → other columns are nulls
- which **unmatched rows do you want to keep**?
- depends on the order in which you join

LEFT [OUTER] JOIN	→	keep unmatched rows in left table
RIGHT [OUTER] JOIN	→	keep unmatched rows in right table
FULL [OUTER] JOIN	→	keep unmatched rows in both left and right tables

Example (left outer join)

T

	j
A	1
B	2
C	

U

k	
1	X
1	Y
	Z

```
SELECT *
FROM T
LEFT JOIN U on T.j = U.k
```

→

A	1	1	X
A	1	1	Y
B	2		
C			

Example (right outer join)

T

	j
A	1
B	2
C	

U

k	
1	X
1	Y
	Z

```
SELECT *
FROM T
RIGHT JOIN U on T.j = U.k
```

A	1	1	X
A	1	1	Y
			Z

Example (full outer join)

T

	j
A	1
B	2
C	

U

k	
1	X
1	Y
	Z

```
SELECT *
FROM T
FULL JOIN U on T.j = U.k
```

→

A	1	1	X
A	1	1	Y
B	2		
C			
			Z

Aggregating

Done last, after projection

Aggregation → **summarize multiple rows into a single row**

- apply an aggregation function for each column

Aggregation functions:

COUNT
MIN
MAX

SUM
AVG

LISTAGG

SELECT COUNT(title), **AVG**(pages) **FROM** Books
→ returns a single row

Aggregating by groups

Can **group** rows together by one or more fields and aggregate **within** each group

```
SELECT COUNT(title), AVG(pages)
FROM Books
GROUP BY year
```

Can put the grouping fields in the SELECT line **without** aggregating them:

```
SELECT COUNT(title), AVG(pages), year
FROM Books
GROUP BY year
```

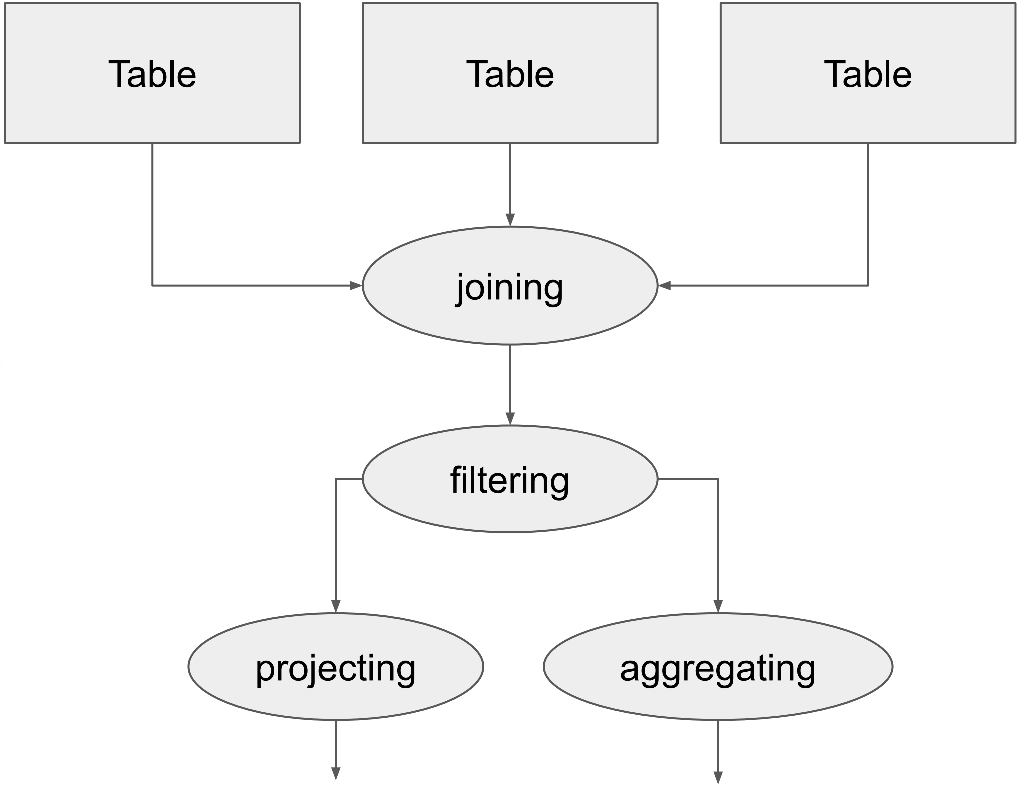
Aggregating works well with joining

```
SELECT Authors.name, COUNT(Books.title), AVG(Books.pages)
FROM Books
JOIN Write USING (isbn)
JOIN Authors USING (authorId)
WHERE Books.year > 1970
GROUP BY Authors.name
```

Can further **filter groups before aggregating** them:

```
…
HAVING COUNT(Books.title) > 2
```

Query flow



Table

Table

Table

joining

filtering

projecting

aggregating

query result rows

That's all, folks!

Appendix

Views	— **CREATE VIEW** *name* **AS** *query*
Distinct as a kind of filtering	— **SELECT DISTINCT** year **FROM** Books
Ordering	— **SELECT** * **FROM** Books **ORDER BY** year, title
Pagination: offset, limit	— **SELECT * FROM** Books **LIMIT** 10
	— **SELECT * FROM** Books **OFFSET** 10 **LIMIT** 10
Table name abbreviations	— **SELECT** B.title **FROM** Books B