Beyond the Relational Algebra: Advanced SQL

Databases

Pierre Senellart

PSL UNIVERSITÉ PARIS

03 March 2026

Outline

DISTINCT

- SQL uses bag semantics by default, the result may contain duplicates

SQL

```
SELECT room
FROM Reservation;
```

- DISTINCT removes duplicates (set semantics)

SQL

```
SELECT DISTINCT room
FROM Reservation;
```

room
504
107
302
504
107

vs

room
107
302
504

ORDER BY

SQL results are unordered unless explicitly sorted. For example, sorting reservations first by most recent date, then by increasing room number:

SQL

```sql
SELECT *
FROM Reservation
ORDER BY arrival DESC, room ASC;
```

- `ORDER BY`, `ORDER BY ... ASC`: sort by ascending value
- `ORDER BY ... DESC`: sort by descending value
- Multiple sort criteria possible
- Sorting is done according to the type: integers are not sorted in the same way as character strings
- Can sort by expressions: e.g.,
  `ORDER BY DATE_ADD(arrival, INTERVAL nights DAY) DESC`

LIMIT and OFFSET

- First two tuples (according to the order of the `ORDER BY` clause)

SQL

```sql
SELECT *
FROM Reservation
ORDER BY arrival
LIMIT 2;
```

- Next two tuples

SQL

```sql
SELECT *
FROM Reservation
ORDER BY arrival
LIMIT 2 OFFSET 2;
```

- Used for pagination (e.g., presenting pages of 10 results at a time)

UNION vs UNION ALL

- Standard UNION is set union, removes duplicates:

SQL

```
SELECT room FROM Reservation
UNION
SELECT room FROM Reservation;
```

- UNION ALL gives multiset union

SQL

```
SELECT room FROM Reservation
UNION ALL
SELECT room FROM Reservation;
```

INTERSECT

- Rooms reserved by guest 2 and on 2026-01-15:

SQL

```
SELECT room FROM Reservation
WHERE guest=2
INTERSECT
SELECT room FROM Reservation
WHERE arrival='2026-01-15';
```

- Returns common tuples, set intersection
- Here, condition can be expressed within a single selection, but not always possible
- Also: `INTERSECT ALL`, multiset intersection

JOIN ... USING

- Shorter syntax for join
- Only when attribute names match (in our example schema: no relevant matching attribute names, so we have to rename attributes to use this)

SQL

```
SELECT Reservation.*, g.name, g.email
FROM Reservation
JOIN (SELECT id AS guest, name, email FROM Guest) AS g
USING(guest);
```

Outer joins

- Joins seen so far are inner joins: they discard tuples without a match
- Outer joins preserve unmatched tuples using NULL values
- Main variants:
  - `LEFT JOIN`: keep left tuples, possibly no right match
  - `RIGHT JOIN`: keep right tuples, possibly no left match
  - `FULL JOIN`: do both (part of SQL standard, but does not exist in MySQL)
- Not expressible in classical relational algebra
- Example: all names of guests, with the room reserved if any

SQL

```
SELECT name, room
FROM Guest
LEFT JOIN Reservation
ON Guest.id = Reservation.guest;
```

Outline

Expressions in SELECT, WHERE, etc.

SQL

```
SELECT id,
       nights,
       nights * 100 AS price
FROM Reservation;
```

- Arithmetic (`+ * - / DIV %`)
- String concatenation: `CONCAT()`
- Date arithmetic
- Many other functions, defined by the DBMS or user-defined

Conditional expressions with CASE

SQL

```sql
SELECT name,
  CASE
    WHEN nights >= 5 THEN 'long stay'
    WHEN nights >= 2 THEN 'short stay'
    ELSE 'one-night stay'
  END AS type
FROM Reservation JOIN Guest
ON guest = Guest.id;
```

- Conditional expressions (similar to `if ... else` in programming languages)
- Evaluated first to last, first match wins
- Returns a value

Handling NULL values with COALESCE

SQL

```sql
SELECT name,
       COALESCE(email, 'unknown')
FROM Guest;
```

- Returns first non-NULL argument
- Useful in outer joins (see further)

NULLIF

- `NULLIF(expr1, expr2)`
- Returns `NULL` if `expr1 = expr2`, otherwise returns `expr1`

SQL

```sql
SELECT name,
       NULLIF(email, '')
FROM Guest;
```

- Equivalent to:

SQL

```sql
CASE
  WHEN expr1 = expr2 THEN NULL
  ELSE expr1
END
```

- Often used to avoid division by zero:

SQL

```sql
price / NULLIF(nights, 0)
```

Outline

Aggregation

Get every room where the average number of nights in a reservation is greater than 3, together with that average number of nights.

SQL

```sql
SELECT room, AVG(nights) AS avg
FROM Reservation
GROUP BY room
HAVING AVG(nights)>3;
```

room	avg
302	6
504	3.5

GROUP BY Semantics

SQL

```
SELECT room, AVG(nights)
FROM Reservation
GROUP BY room;
```

- Partitions rows into groups
- One output row per group
- Every selected attribute must be either:
  - grouped (within `GROUP BY`)
  - or aggregated (with an aggregate function)
- ... at least in theory, MySQL does not enforce the rule, which results in some unpredictable results

DISTINCT vs GROUP BY

- Removing duplicates: two equivalent ways

SQL

```sql
SELECT DISTINCT room
FROM Reservation;
```

SQL

```sql
SELECT room
FROM Reservation
GROUP BY room;
```

- When selecting only grouped attributes, `GROUP BY` behaves like `DISTINCT`
- `GROUP BY` is more general: it allows aggregation (e.g., `COUNT(*)`)
- Conceptually:
  - `DISTINCT` = duplicate elimination
  - `GROUP BY` = partition into groups

Common SQL aggregate functions

Function	Meaning
`COUNT(*)`	number of rows in the group
`COUNT(expr)`	number of non-NULL values of expr
`COUNT(DISTINCT expr)`	number of distinct non-NULL values of expr
`SUM(expr)`	sum of non-NULL values
`AVG(expr)`	average of non-NULL values
`MIN(expr)`	minimum non-NULL value
`MAX(expr)`	maximum non-NULL value
`GROUP_CONCAT(expr)`	concatenate values into a string (specific to MySQL)

WHERE vs HAVING

SQL

```sql
SELECT room, AVG(nights)
FROM Reservation
WHERE arrival > '2026-01-01'
GROUP BY room
HAVING AVG(nights) > 3;
```

- `WHERE` filters rows before grouping
- `HAVING` filters groups after aggregation

HAVING without GROUP BY

- If there is no `GROUP BY`, the entire table forms a single group

SQL

```sql
SELECT COUNT(*) AS total
FROM Reservation
HAVING COUNT(*) > 10;
```

- Filters the single group after aggregation
- Equivalent to:

SQL

```sql
SELECT *
FROM (
  SELECT COUNT(*) AS total
  FROM Reservation
) t
WHERE total > 10;
```

Window functions

- Window functions perform aggregations without collapsing rows
- Defined over a window of related tuples
- As they depend on ordering, not expressible in classical relational algebra
- Compute an aggregation over a whole group, or (if there is an `ORDER BY`) over the window formed of every element of a group up to the current element

SQL

```
f(expr) OVER (
  PARTITION BY ...
  ORDER BY ...
)
```

- Example: average number of nights per room, shown for each reservation

SQL

```
SELECT room, nights,
       AVG(nights) OVER (PARTITION BY room) AS avg
FROM Reservation;
```

Ranking with Window Functions

SQL

```sql
SELECT room,
       nights,
       RANK() OVER (
         PARTITION BY room
         ORDER BY nights DESC
       ) AS ranking
FROM Reservation;
```

- No row collapse
- One of the main uses of window functions

Outline

Nested queries

- In the `FROM` clause, a query (with an alias) can be used as a regular table
- SQL also allows queries inside the `WHERE` clause
- Correspond to existential / universal quantification
- Do not increase expressive power beyond relational algebra, but provide more convenient syntax
- Two main forms:
  - `IN`, `NOT IN`
  - `EXISTS`, `NOT EXISTS`
- Example: names of guests with at least one reservation

SQL

```
SELECT name
FROM Guest
WHERE EXISTS (
  SELECT *
  FROM Reservation r
  WHERE r.guest = Guest.id
);
```

IN and EXISTS

SQL

```sql
SELECT name
FROM Guest
WHERE id IN (
  SELECT guest FROM Reservation
);
```

same as:

SQL

```sql
SELECT name
FROM Guest
WHERE EXISTS (
  SELECT *
  FROM Reservation r
  WHERE r.guest = Guest.id
);
```

NOT IN vs NOT EXISTS

SQL

```sql
SELECT name
FROM Guest
WHERE id NOT IN (
  SELECT guest FROM Reservation
);
```

SQL

```sql
SELECT name
FROM Guest
WHERE NOT EXISTS (
  SELECT *
  FROM Reservation r
  WHERE r.guest = Guest.id
);
```

NOT IN: Caveat

- If the subquery of `NOT IN` contains a `NULL`, the result is empty (three-valued logic)
- `NOT EXISTS` does not have this issue
- In practice: prefer `NOT EXISTS`

SQL

```
CREATE TABLE Test  (x INT);
CREATE TABLE TestN (x INT);
INSERT INTO Test  VALUES (1), (2), (3);
INSERT INTO TestN VALUES (1), (2), (NULL);

SELECT * FROM Test
WHERE x NOT IN (SELECT x FROM TestN);
SELECT * FROM Test
WHERE NOT EXISTS (SELECT x FROM TestN WHERE Test.x=TestN.x);
```

Correlated vs non-correlated subqueries

- Non-correlated: independent of outer query; can be evaluated once

SQL

```sql
SELECT name
FROM Guest
WHERE id IN (
  SELECT guest FROM Reservation
);
```

- Correlated: refers to outer query variables; logically evaluated once per outer row, may be optimized

SQL

```sql
SELECT name
FROM Guest g
WHERE EXISTS (
  SELECT *
  FROM Reservation r
  WHERE r.guest = g.id
);
```

Scalar subqueries

- A subquery that returns a single value
- Can be used wherever an expression is allowed

SQL

```sql
SELECT name
FROM Guest
WHERE id = (
  SELECT guest
  FROM Reservation
  ORDER BY arrival DESC
  LIMIT 1
);
```

- Must return at most one row
- Otherwise: runtime error
- Behaves like a constant value in the outer query
- If no row returned: NULL

Outline

Common Table Expressions (CTEs)

- Defines a temporary named query
- Improves readability and modularity
- Equivalent to using a subquery in the `FROM` clause

SQL

```
WITH LongStays AS (
  SELECT *
  FROM Reservation
  WHERE nights >= 5
)
SELECT room, COUNT(*)
FROM LongStays
GROUP BY room;
```

- Scope limited to the query
- Non-recursive form (recursive form uses `WITH RECURSIVE`)

Named views

- A view is a named query stored in the database
- Defined once, reused in multiple queries
- Improves modularity and abstraction

SQL

```sql
CREATE VIEW LongStays AS
SELECT *
FROM Reservation
WHERE nights >= 5;
```

SQL

```sql
SELECT room, COUNT(*)
FROM LongStays
GROUP BY room;
```

- Unlike a CTE:
  - persists in the database schema
  - can be used by any query
- Usually behaves like a virtual table (no data stored)

Outline

Logical Query Processing Order

SQL is declarative but logically evaluated following a given order:

1. `FROM`
2. `WHERE`
3. `GROUP BY`
4. `HAVING`
5. `SELECT`
6. `DISTINCT`
7. `ORDER BY`
8. `LIMIT`

Example consequence: you can use column aliases (`AS`) introduced in a `SELECT` clause in a `ORDER BY` clause but not in a `WHERE` clause

References

- Classic textbook on database systems [Garcia-Molina et al., 2008]
- More recent one, also very complete [Silberschatz et al., 2019]
- Classic textbook on database theory [Abiteboul et al., 1995] (advanced material, but the first chapters are readable)
- Details of SQL: standards are not public documents (and not useful for a final user); use the RDBMS documentation instead
- For example, MySQL has a comprehensive documentation at `https://dev.mysql.com/doc/refman/9.6/en/sql-statements.html` (and using \h in the command line client)

Licensing

Bibliography I

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. http://webdam.inria.fr/Alice/.

Hector Garcia-Molina, Jeffrey D. Ulman, and Jennifer Widom. *Database Systems: The Complete Book*. Pearson, 2nd edition, 2008.

Avi Silberschatz, Henry F. Korth, and S. Sudarshan. Database system concepts, 2019.