链滴

ANSI SQL 及实现调研 (一)

作者：flowaters

原文链接：https://ld246.com/article/1528183787124

来源网站：链滴

背景

- 经常看到SQL-92，SQL-03的词汇，但是不了解其中的差异，比如SQL-03比SQL-92多了些什么?
- 知道不同的数据库有不同的方言，但是不了解哪些是SQL标准中定义的大家都遵守的。
- 开发一个类SQL语言时，选择支持哪些语法好呢?

本文从ANSI SQL切入，来调研一下ANSI SQL标准(下面简称为SQL标准)的演变，以及不同数据库对SL标准的支持程度。

历程

通过SQL Wikipedia上了解到：

由于不同的数据库产商的SQL语法不兼容，因此1986年开始，ANSI和ISO标准化组织采纳了SQL语言准。并且在随后的年份陆续公布了新的标准。

具体年份和名称如下：

年份	名称	备注
1986	SQL-86	第一版
1989	SQL-89	小修改
1992	SQL-92	大修改
1999	SQL:1999	加入嵌套表(nested table[4]), 加入正则表达式, 递归查询，触发器，过程和控制流，数组类型和一些结构化类型
2003	SQL:2003	加入XML相关征，窗口函数，标准化序列(standardized sequences)，自动生成值的列
2006	SQL:2006	支持导入，存和操作XML数据，加入XQuery
2008	SQL:2008	支持cursor定外的ORDER BY?, 增加INSTEAD OF触发器，增加TRUNCATE，增加FETCH
2011	SQL:2011	增加temporal data，增强window functions和FETCH
2016	SQL:2016	增加列特征匹配(ow pattern matching), 多态表函数(polymorphic table functions), JSON

标准公开出来后，不同的产商的实现情况也是不兼容的，即不完全遵守标准。主要是日期类型，时间型，字符串拼接，NULL和大小写敏感。

只有PostgreSQL和Mimer SQL是紧跟标准的。

变化

请找一个充足的时间，打开Modern SQL: Slides，从头到尾看一遍。

或者打开Modern SQL in Open Source and Commercial Databases，下载下来看。

这里做一下读书笔记

LATERAL

来自SQL:1999

依旧拿PostgreSQL初体验中的示例数据举例，现在有两张表cities和weather如下：

```
mydb=# select * from cities;
     name      | location
---------------+-----------
 San Francisco | (-194,53)
(1 row)

mydb=# select * from weather;
     city      | temp_lo | temp_hi | prcp |    date
---------------+---------+---------+------+------------
 San Francisco |      46 |      50 | 0.25 | 1994-11-27
 San Francisco |      43 |      57 |    0 | 1994-11-29
 Hayward       |      37 |      54 |      | 1994-11-29
(3 rows)
```

JOIN LATERAL

现在两张表要做一个JOIN

在SQL:1999之前，只能这么写：

```
SELECT *
FROM cities
    JOIN (
        SELECT *
        FROM weather
    ) inline_view
    ON inline_view.city = cities.name
```

执行结果

```
     name      | location  |     city      | temp_lo | temp_hi | prcp |    date
---------------+-----------+---------------+---------+---------+------+------------
 San Francisco | (-194,53) | San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco | (-194,53) | San Francisco |      46 |      50 | 0.25 | 1994-11-27
(2 rows)
```

在SQL:1999之后，可以这样写：

```
SELECT *
FROM cities
    JOIN lateral(
        SELECT *
        FROM weather
        WHERE weather.city = cities.name
    ) inline_view ON true
```

结果

```
     name      | location |     city      | temp_lo | temp_hi | prcp |    date
---------------+----------+---------------+---------+---------+------+------------
 San Francisco | (-194,53) | San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco | (-194,53) | San Francisco |      46 |      50 | 0.25 | 1994-11-27
(2 rows)
```

注意两点：

- 在 inline_view中，引用了外部的cities
- JOIN使用了join lateral关键字，而不仅仅是join

好奇一下内部的执行过程是不是一样的，使用explain来看一下

```
EXPLAIN SELECT *
FROM cities
   JOIN (
      SELECT *
      FROM weather
   ) inline_view
   ON inline_view.city = cities.name;
```

结果

```
                        QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=18.10..55.28 rows=648 width=388)
   Hash Cond: ((cities.name)::text = (weather.city)::text)
   ->  Seq Scan on cities  (cost=0.00..13.60 rows=360 width=194)
   ->  Hash  (cost=13.60..13.60 rows=360 width=194)
         ->  Seq Scan on weather  (cost=0.00..13.60 rows=360 width=194)
(5 rows)

EXPLAIN SELECT *
FROM cities
   JOIN lateral(
      SELECT *
      FROM weather
      WHERE weather.city = cities.name
   ) inline_view ON true;
```

结果

```
                        QUERY PLAN
-----------------------------------------------------------------------
 Hash Join  (cost=18.10..55.28 rows=648 width=388)
   Hash Cond: ((cities.name)::text = (weather.city)::text)
   ->  Seq Scan on cities  (cost=0.00..13.60 rows=360 width=194)
   ->  Hash  (cost=13.60..13.60 rows=360 width=194)
         ->  Seq Scan on weather  (cost=0.00..13.60 rows=360 width=194)
(5 rows)
```

这两个的实际执行过程是一样的。

CROSS JOIN LATERAL

使用 cross join lateral也可以实际同样的效果，同时可以不加末尾的 on true

```
SELECT *
FROM cities
   CROSS JOIN lateral(
      SELECT *
      FROM weather
      WHERE weather.city = cities.name
   ) inline_view;
```

结果

```
     name      | location  |     city      | temp_lo | temp_hi | prcp |    date
---------------+-----------+---------------+---------+---------+------+------------
 San Francisco | (-194,53) | San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco | (-194,53) | San Francisco |      46 |      50 | 0.25 | 1994-11-27
(2 rows)
```

内部执行过程仍然是相同的

```
EXPLAIN SELECT *
FROM cities
   CROSS JOIN lateral(
      SELECT *
      FROM weather
      WHERE weather.city = cities.name
   ) inline_view;
```

结果

```
                            QUERY PLAN
-------------------------------------------------------------------------------
 Hash Join  (cost=18.10..55.28 rows=648 width=388)
   Hash Cond: ((cities.name)::text = (weather.city)::text)
   ->  Seq Scan on cities  (cost=0.00..13.60 rows=360 width=194)
   ->  Hash  (cost=13.60..13.60 rows=360 width=194)
         ->  Seq Scan on weather  (cost=0.00..13.60 rows=360 width=194)
(5 rows)
```

意义

这种写法有什么意义呢？比原来的写法的优越性在哪里呢？

一方面是 LATERAL和table functions可以联合使用。这个目前是PostgreSQL中特有的，本文略过。

LATERAL & LIMIT

另一方面是 可以在 inline view中加入LIMIT操作，比如：

```
SELECT *
FROM cities
```

```
    CROSS JOIN lateral(
        SELECT *
        FROM weather
        WHERE weather.city = cities.name
        ORDER BY temp_lo
        LIMIT 1
    ) inline_view;
```

结果

```
     name      | location  |     city      | temp_lo | temp_hi | prcp |    date
---------------+-----------+---------------+---------+---------+------+------------
 San Francisco | (-194,53) | San Francisco |      43 |      57 |    0 | 1994-11-29
(1 row)
```

如果不使用LATERAL的话，是无法实现只取某一列的前LIMIT行的。

LATERAL & Multi-Source Top-N

场景：用户找出每个分类下最新的10条新闻。

```
SELECT n.*
FROM news n
    JOIN subscriptions s
    ON n.topic = s.topic
WHERE s.user = ?
ORDER BY n.created DESC
LIMIT 10
```

这样会把每个subscriptions下的所有的news都找出来，但是是没有必要的。

使用LATERAL的写法如下：

```
SELECT n.*
FROM subscriptions s
    JOIN LATERAL(
        SELECT *
        FROM news n
        WHERE n.topic = s.topic
        ORDER BY n.created DESC
        LIMIT 10
    ) top_news ON true
WHERE s.user_id = ?
ORDER BY n.created DESC
LIMIT 10;
```

这样可以做到只取每个news下的前10条新闻.

小结

- SQL中的"FOR EACH"
- 适合与OUTER JOINS联用
- 适合取子查询中的Top-N

- 可以与table functions作join

支持的数据库

- DB2 LUW 9.1+
- Oracle 12c+
- PostgreSQL 9.3+
- SQL Server 2005部分支持
- MySQL不支持
- SQLite不支持

WITH (Common Table Expressions)

来自SQL:1999

WITH是解决什么问题的呢?

背景

OK，先来看一个在SQL:1999之前的场景：

当遇到多层嵌套的SQL时，我们会这样写。

```
SELECT ...
  FROM (SELECT ...
          FROM t1
          JOIN (SELECT ... FROM ...
              ) a ON (...)
       ) b
  JOIN (SELECT ... FROM ...
       ) c ON (...)
```

下面的SQL是为了示例而示例：

```
SELECT *
FROM (
    SELECT *
    FROM cities
) a
    JOIN (
        SELECT *
        FROM weather
    ) b
    ON a.name = b.city;
```

结果

```
     name      | location  |     city      | temp_lo | temp_hi | prcp |    date
---------------+-----------+---------------+---------+---------+------+------------
 San Francisco | (-194,53) | San Francisco |      43 |      57 |    0 | 1994-11-29
```

```
 San Francisco | (-194,53) | San Francisco |      46 |      50 | 0.25 | 1994-11-27
(2 rows)
```

当去理解这样的SQL语句时，需要先理解最里面的SELECT，再理解次里面的SELECT，一层一层的向看，最后再理解最外层的SELECT。在SQL92时，就是这样来做的。

如果有了WITH CTEs，就可以用更易理解的方式来写SQL了。

语法

一个CTE

```
WITH
  a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM ...)
SELECT ...
```

两个CTEs

```
WITH
  a (c1, c2, c3)
AS (SELECT c1, c2, c3 FROM ...),
  b (c4, ...)
AS (SELECT c4, ...
    FROM t1
    JOIN a
      ON(...)
   )

SELECT ...
```

- 多个CTE之间由逗号连接，只有第一个CTE前有WITH关键字，后面的CTE不需要WITH关键字。
- 在第二个CTE中可以引用第一个CTE，即在后面的CTE可以引用前面的CTE。
- 最后一个CTE后没有逗号，表明后面跟着的是查询语句。

这样的话，看SQL时就可以从上往下来看了。

将上面的示例重写一下：

```
WITH a (name, location) AS (
        SELECT name, location
        FROM cities
    ),
    b (city, temp_lo, temp_hi, prcp, date) AS (
        SELECT city, temp_lo, temp_hi, prcp, date
        FROM weather
    )
SELECT *
FROM a
    JOIN b ON a.name = b.city;
```

结果如下：

```
     name      | location |     city      | temp_lo | temp_hi | prcp |    date
---------------+----------+---------------+---------+---------+------+------------
 San Francisco | (-194,53) | San Francisco |      43 |      57 |    0 | 1994-11-29
 San Francisco | (-194,53) | San Francisco |      46 |      50 | 0.25 | 1994-11-27
(2 rows)
```

小结

- WITH是SQL中的"私有方法 private methods"
- WITH视图可以多次被引用
- WITH通过定义多个CTE，来实现嵌套
- 可以用SELECT的地方，就可以用WITH，如 INSERT INTO tbl WITH ... SELECT

坑

下推(push down)

需要注意的是，在PostgreSQL中，WITH视图更像是一个物化视图，没有把查询中的过滤条件下推到图中。

在数据量大时，可能会有性能问题。使用时请用EXPLAIN来仔细检查一下。

但是在PostgreSQL中使用inline视图时，会有下推的操作的。

上推(push up)

PostgreSQL 9.1+中，INSERT, UPDATE和DELETE中也支持WITH。

如将指定行从一个表移动到另一个表

```
WITH deleted_rows AS(
  DELETE FROM source_tbl
  RETURNING *
)
INSERT INTO destination_tbl
SELECT * FROM deleted_rows
```

支持的数据库

- DB2 LUW 7+
- Oracle 9iR2+
- PostgreSQL 8.4+
- SQL Server 2005+
- SQLite 3.8.3+
- MySQL 8.0+

WITH RECURSIVE (Common Table Expressions)

来自SQL:1999

从一个例子来看起。

生成1到3的数字

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
      UNION ALL
     SELECT n+1
       FROM cte
      WHERE n < 3)
SELECT * FROM cte;
```

结果

```
 n
---
 1
 2
 3
(3 rows)
```

这个是怎么实现的呢?

首先其中的SELECT中有两项，由UNION (ALL)分开: 初始项和递归项，见 SELECT 初始项 UNION ALL 递归项

初始时，由初始项产生临时表

随时，临时表做为递归项的输入表，再生产新的临时表

直到新的临时表为空时结束

这样，就形成了一个循环。

来看更多的例子

生成2到4的数字

```
WITH RECURSIVE cte (n)
  AS (SELECT 2
      UNION ALL
     SELECT n + 1
       FROM cte
      WHERE n < 4)
SELECT * FROM cte;
```

结果

```
 n
---
 2
 3
 4
```

```
(3 rows)
```

生成1,3,5的数字

```
WITH RECURSIVE cte (n)
  AS (SELECT 1
       UNION ALL
      SELECT n + 2
        FROM cte
      WHERE n < 5)
SELECT * FROM cte;
```

结果

```
 n
---
 1
 3
 5
(3 rows)
```

阶乘

再看一个求阶乘的例子

```
WITH RECURSIVE factorial(F,n) AS (
  SELECT 1 F, 4 n
UNION ALL
  SELECT F*n F, n-1 n
  FROM factorial
  WHERE n > 1)
SELECT * FROM factorial;
```

结果

```
 f  | n
----+---
  1 | 4
  4 | 3
 12 | 2
 24 | 1
(4 rows)
```

即4的阶乘为24。

求树的子节点

构造一颗树，每一个节点有自己编号id, 和其父节点的编号parent_id

建表

```
CREATE TABLE t (
  id NUMERIC NOT NULL,
```

```
  parent_id NUMERIC,
  PRIMARY KEY (id)
)
```

构造数据

```
INSERT INTO t VALUES (1, null);

INSERT INTO t VALUES (2, 1);
INSERT INTO t VALUES (3, 1);
INSERT INTO t VALUES (4, 1);
INSERT INTO t VALUES (5, 1);

INSERT INTO t VALUES (6, 2);
INSERT INTO t VALUES (7, 3);
INSERT INTO t VALUES (8, 3);
INSERT INTO t VALUES (9, 4);
INSERT INTO t VALUES (10, 5);

INSERT INTO t VALUES (11, 6);
INSERT INTO t VALUES (12, 8);
INSERT INTO t VALUES (13, 8);
INSERT INTO t VALUES (14, 8);
INSERT INTO t VALUES (15, 8);
INSERT INTO t VALUES (16, 10);
```

查看一下表中的数据

```
mydb=# select * from t;
 id | parent_id
----+-----------
  1 |
  2 |         1
  3 |         1
  4 |         1
  5 |         1
  6 |         2
  7 |         3
  8 |         3
  9 |         4
 10 |         5
 11 |         6
 12 |         8
 13 |         8
 14 |         8
 15 |         8
 16 |        10
(16 rows)
```

现在来找到节点3其及所有子节点。

传统写法

可以使用多个LEFT JOIN来写成，同时自己的结果再进行二次处理。

```
SELECT d0.*, d1.*, d2.*
  FROM t AS d0
  LEFT JOIN t AS d1
    ON (d1.parent_id = d0.id)
  LEFT JOIN t AS d2
    ON (d2.parent_id = d1.id)
  WHERE d0.id = 3
```

结果如下：

```
 id | parent_id | id | parent_id | id | parent_id
----+-----------+----+-----------+----+-----------
  3 |         1 |  8 |         3 | 12 |         8
  3 |         1 |  8 |         3 | 13 |         8
  3 |         1 |  8 |         3 | 14 |         8
  3 |         1 |  8 |         3 | 15 |         8
  3 |         1 |  7 |         3 |    |
(5 rows)
```

WITH RECURSIVE CTEs写法

```
WITH RECURSIVE subtree (id, parent_id) AS
(
        SELECT id, parent_id
        FROM t
        WHERE id = 3
    UNION ALL
        SELECT t.id, t.parent_id
        FROM subtree
        LEFT JOIN t
            ON (t.parent_id = subtree.id)
        WHERE t.id <= 20
)
SELECT * FROM subtree;
```

结果

```
 id | parent_id
----+-----------
  3 |         1
  7 |         3
  8 |         3
 12 |         8
 13 |         8
 14 |         8
 15 |         8
(7 rows)
```

小结

- SQL中的while循环
- 列生成器

- 图处理器
- 可以将数据传递到下一个迭代中
- 需要有终止条件，否则容易死循环

支持的数据库

- DB2 LUV 7+
- Oracle 11.gR2+
- PostgreSQL 8.4+
- SQL Server 2005+
- SQLite 3.8.3+
- MySQL 不支持

OVER/PARTITION BY

来自 SQL:2003

先从一个场景来说起，如何实现部分收入百分比呢?

构造数据

```
CREATE TABLE empsalary(
  depname varchar,
  empno bigint,
  salary int,
  enroll_date date
);

INSERT INTO empsalary VALUES('develop',10, 5200, '2007/08/01');
INSERT INTO empsalary VALUES('sales', 1, 5000, '2006/10/01');
INSERT INTO empsalary VALUES('personnel', 5, 3500, '2007/12/10');
INSERT INTO empsalary VALUES('sales', 4, 4800, '2007/08/08');
INSERT INTO empsalary VALUES('sales', 6, 5500, '2007/01/02');
INSERT INTO empsalary VALUES('personnel', 2, 3900, '2006/12/23');
INSERT INTO empsalary VALUES('develop', 7, 4200, '2008/01/01');
INSERT INTO empsalary VALUES('develop', 9, 4500, '2008/01/01');
INSERT INTO empsalary VALUES('sales', 3, 4800, '2007/08/01');
INSERT INTO empsalary VALUES('develop', 8, 6000, '2006/10/01');
INSERT INTO empsalary VALUES('develop', 11, 5200, '2007/08/15');
```

SQL:2003之前解法

在SQL:2003之前

```
WITH total_salary_by_department
  AS (SELECT depname, SUM(salary) total
      FROM empsalary
      GROUP BY depname)
```

```
SELECT ts.depname, empno, salary,
  100 * salary/ts.total  "% of dep"
  FROM empsalary
  JOIN total_salary_by_department ts
    ON (empsalary.depname = ts.depname)
  ORDER BY depname, "% of dep" desc;
```

结果

```
 develop   |     8 |   6000 |       23
 develop   |    10 |   5200 |       20
 develop   |    11 |   5200 |       20
 develop   |     9 |   4500 |       17
 develop   |     7 |   4200 |       16
 personnel |     2 |   3900 |       52
 personnel |     5 |   3500 |       47
 sales     |     6 |   5500 |       27
 sales     |     1 |   5000 |       24
 sales     |     3 |   4800 |       23
 sales     |     4 |   4800 |       23
```

如果只看某一个部门的呢?

```
WITH total_salary_by_department
  AS (SELECT depname, SUM(salary) total
      FROM empsalary
      GROUP BY depname)
SELECT ts.depname, empno, salary,
  100 * salary/ts.total  "% of dep"
  FROM empsalary
  JOIN total_salary_by_department ts
    ON (empsalary.depname = ts.depname)
  WHERE empsalary.depname = 'develop'
  ORDER BY depname, "% of dep" desc;
```

结果

```
 develop |     8 |   6000 |       23
 develop |    10 |   5200 |       20
 develop |    11 |   5200 |       20
 develop |     9 |   4500 |       17
 develop |     7 |   4200 |       16
```

好吧，这个又是上面提到的PostgreSQL的性能问题。

在SQL:2003以前，想实现Aggregation的话，只有通过DISTINCT和GROUP BY这两种方法。如果不选项被合并的话，那就只有GROUP BY一种方法。

SQL:2003

在SQL:2003中，引入了OVER(PARTITION BY)语法，来解法此类问题。

如上面的SQL，可以写为：

```
SELECT depname,
        empno,
        salary,
        100 * salary /  SUM(salary)
              OVER(PARTITION BY depname) "% of depname"
       FROM empsalary
       ORDER BY
         depname,
         "% of depname" DESC;
```

结果

```
 develop   |     8 |   6000 |          23
 develop   |    10 |   5200 |          20
 develop   |    11 |   5200 |          20
 develop   |     9 |   4500 |          17
 develop   |     7 |   4200 |          16
 personnel |     2 |   3900 |          52
 personnel |     5 |   3500 |          47
 sales     |     6 |   5500 |          27
 sales     |     1 |   5000 |          24
 sales     |     3 |   4800 |          23
 sales     |     4 |   4800 |          23
```

这里先选择了salary列，然后选择对salary列做SUM，同时通过PARTIITON BY关键字指定了SUM的口是针对dep列。

小结

- OVER可以与任何聚合函数搭配作用
- 略
- 略
- OVER (PARTITION BY X)可以实现类似GROUP BY的效果

OVER/ORDER BY

收支场景：

有一张表transactions记录着自己的收支情况。现在需要显示收支的明细和对应的余额。

构造数据

```
CREATE TABLE transactions(
  txid int,
  value numeric
);

INSERT INTO transactions VALUES(1, +10);
INSERT INTO transactions VALUES(2, +20);
INSERT INTO transactions VALUES(3, -10);
INSERT INTO transactions VALUES(4, +50);
```

```
INSERT INTO transactions VALUES(5, -30);
INSERT INTO transactions VALUES(6, -20);
```

查看如下

```
mydb=# select * from transactions;
 1 |  10
 2 |  20
 3 | -10
 4 |  50
 5 | -30
 6 | -20
```

SQL:2003之前

```
SELECT txid,
       value,
       (SELECT SUM(value)
          FROM transactions tx2
       WHERE tx2.txid <= tx1.txid) balance
   FROM transactions tx1
   ORDER BY txid;
```

结果

```
   1 |  10 |  10
   2 |  20 |  30
   3 | -10 |  20
   4 |  50 |  70
   5 | -30 |  40
   6 | -20 |  20
```

SQL:2003

```
SELECT txid,
       value,
       SUM(value)
       OVER(ORDER BY txid
          ROWS
          BETWEEN UNBOUNDED PRECEDING
              AND CURRENT ROW
          ) balance
   FROM transactions tx1
   ORDER BY txid;
```

结果

```
   1 |  10 |  10
   2 |  20 |  30
   3 | -10 |  20
   4 |  50 |  70
   5 | -30 |  40
   6 | -20 |  20
```

新的方法

- ROW_NUMBER

排序方法

- RANK
- DENSE_RANK
- PERCENT_RANK
- CUME_DIST

支持的数据库

- DB2 LUW: 7+
- Oracle: 8i+
- PostgreSQL: 8.4+
- SQL Server: 2005+
- MySQL: 不支持
- SQLite: 不支持

OVER/LAG

SQL:2008

场景：显示与前一项的差值。和上面的场景正好相反。

SQL:2008前

显示ROW_NUMBER

```
 SELECT *, ROW_NUMBER() OVER (ORDER BY txid) rn FROM transactions;
   1 |    10 | 1
   2 |    20 | 2
   3 |   -10 | 3
   4 |    50 | 4
   5 |   -30 | 5
   6 |   -20 | 6
```

然后使用上面提到的SQL:2003的WITH方法，来求差值。

```
WITH numbered_data AS (
  SELECT *,
      ROW_NUMBER() OVER (ORDER BY txid) rn
      FROM transactions
)
SELECT cur.txid, cur.value, cur.value - prev.value
  FROM       numbered_data cur
```

```
  LEFT JOIN numbered_data prev
    ON (cur.rn - 1 = prev.rn);
```

结果

```
 1 |    10 |
 2 |    20 |       10
 3 |   -10 |      -30
 4 |    50 |       60
 5 |   -30 |      -80
 6 |   -20 |       10
```

SQL:2008

```
SELECT *,
       value - LAG(value)
             OVER (ORDER BY txid)
      FROM transactions;
```

结果：

```
 1 |    10 |
 2 |    20 |       10
 3 |   -10 |      -30
 4 |    50 |       60
 5 |   -30 |      -80
 6 |   -20 |       10
```

其它

同样的还有：

```
  LEAD / LAG
  FIRST_VALUE / LAST_VALUE
  MIN_VALUE(col, n) FROM FIRST/LAST
                    RESPECT/IGNORE NULLS
```

支持的数据库

- DB2 LUW 9.5+
- Oracle: 11gR2+
- PostgreSQL 8.4+
- SQL Server 2012+
- MySQL: 不支持

FETCH FIRST

SQL:2008

场景：找到选定行的前有限列

SQL:2008之前

```
SELECT *
  FROM (SELECT *,
          ROW_NUMBER() OVER (ORDER BY txid) rn
        FROM transactions) numbered_data
  WHERE rn <= 3;
```

结果

```
   1 |    10 | 1
   2 |    20 | 2
   3 |   -10 | 3
```

当然也可以使用非标准的语法，如LIMIT, TOP。

SQL:2008之后

```
SELECT *
    FROM transactions
  ORDER BY txid
  FETCH FIRST 3 ROWS ONLY;
```

结果

```
   1 |    10
   2 |    20
   3 |   -10
```

支持的数据库

- DB2 LUW 7+
- MySQL 3.19.3+
- Oracle 12c
- PostgreSQL 8.4+
- SQL Server 2012+
- SQLite 2.1.0+

OFFSET

SQL: 2011

这个很常见了，但是只是在SQL:2011中才引入。

支持的数据库

- DB2 LUW 9.7+
- MySQL 4.0.6+

- Oracle 12c+
- PostgreSQL 6.5+
- SQL Server 2012+
- SQLite 2.1.0+

AS OF

场景：记录数据记录的产生时间

SQL:2011中

```
CREATE TABLE t( ...,
    start_ts TIMESTAMP(9) GENERATED
            ALWAYS AS ROW START,
    end_ts   TIMESTAMP(9) GENERATED
            ALWAYS AS ROW END,
    PERIOD FOR SYSTEM TIME (start_ts, end_ts)
) WITH SYSTEM VERSIONING
```

写入

```
INSERT ... (ID, DATA) VALUES (1, 'X')
```

SQL:2011

其中的start_ts和end_ts会自动修改，对应用透明。

支持的数据库

- DB2 LUW 10.1+
- Oracle 10gR1+
- MySQL 不支持
- PostgreSQL 不支持
- SQL Server 不支持
- SQLite 不支持

WITHOUT OVERLAPS

场景：每行日志有起止时间，要求约束起止时间不重叠。

SQL:2011

SQL:2011中引入了temporal类型和bi-temporal类型，如：

```
PRIMARY KEY (id, period WITHOUT OVERLAPS)
```

支持的数据库

- DB2 LUW 10.1+
- PostgreSQL 9.2+
- MySQL 不支持
- Oracle 不支持
- SQL Server 不支持
- SQLite 不支持

参考

1. One Giant Leap For SQL: MySQL 8.0 Released
2. SQL Wikipedia
3. Modern SQL: Slides
4. Modern SQL: Video
5. Comparison of different SQL implementations
6. How to select using with recursive clause
7. Early History of SQL
8. What's new in SQL:2011
9. What's New in SQL:2016
10. sql在线美化/格式化/压缩