蚂蚁金服金融科技产品手册

SQL参考（Oracle）

产品版本： 2.0.0

文档版本： V20200416

蚂蚁金服金融科技文档

商标声明

及其他蚂蚁金服金融科技服务相关的商标均为蚂蚁金服金融科技所有。
本文档涉及的第三方的注册商标，依法由权利人所有。

免责声明

由于产品版本升级、调整或其他原因，本文档内容有可能变更。蚂蚁金服金融科技保留在没有任何通知或者提示下对本文档的内容进行修改的权利，并在蚂蚁金服金融科技授权通道中不时发布更新后的用户文档。您应当实时关注用户文档的版本变更并通过蚂蚁金服金融科技授权渠道下载、获取最新版的用户文档。如因文档使用不当造成的直接或间接损失，本公司不承担任何责任。

目录

1 与 Oracle 兼容性对比

OceanBase 支持 Oracle 中绝大部分的基本 SQL 语法。这意味着在从 Oracle 过渡到 OceanBase 的过程中，您不需要消耗大量的时间去学习新的语法，并且可以流畅的实现从 Oracle 到 OceanBase 的迁移。

基于优化和开发的考虑，有一些功能 OceanBase 暂不支持或者是与 Oracle 的表现有所差异。所以本篇文档中将分节对比 OceanBase 对 Oracle 的兼容。

SQL 语法

- OceanBase 支持 Oracle 中基本 SQL 语法。
- 少数功能性缺失会报语法不支持的错误，例如在层次查询包含多表连接的场景下就会报错。
- OceanBase 暂不支持 Oracle 部分较复杂的 OLAP 语法，如模式匹配、 PIVOT/ UNPIVOT 函数、多态表函数和频繁项目集计算。

SQL 数据类型

- Oracle 中有 24 个数据类型， OceanBase 目前支持 18 种，详细信息请参阅章节 内建数据类型 。
- 基于优化的考虑，LONG 和 LONG RAW 数据类型过于老旧，所以 OceanBase 不计划支持这两种数据类型。
- OceanBase 中大对象数据类型有 48 M 的大小限制且性能不佳，所以不推荐在复杂场景下使用。详细信息请参阅章节 大对象类型 。

字符集和 Collation

- OceanBase 支持 UTF-8、UTF-16、GBK、GB18030 和国家字符集。
- OceanBase 仅支持大小写敏感的 Collation。
- OceanBase 不支持多语言语义的 Collation 比较和排序。

内建函数

Oracle 中支持内建函数 117 个，OceanBase 暂时支持 103 个，详细信息请参阅章节 函数 。

系统视图

- Oracle 中有字典视图 400 多个，OceanBase 目前兼容 17 个：
    - ALL_CONS_COLUMNS
    - ALL_CONSTRAINTS
    - ALL_IND_COLUMNS
    - ALL_INDEX
    - ALL_OBJECTS
    - ALL_PART_KEY_COLUMNS
    - ALL_PART_TABLES
    - ALL_SEQUENCE
    - ALL_SOURCE

        - ALL_SUBPART_KEY_COLUMNS
        - ALL_SYNONYMS
        - ALL_TAB_COLUMNS
        - ALL_TAB_PARTITIONS
        - ALL_TABLES
        - ALL_TYPES
        - ALL_USERS
        - ALL_VIEWS
    - Oracle 中有性能视图 700 多个，OceanBase 目前兼容 3 个。
        - V$SYSTEM_EVENT
        - V$SESSION_WAIT
        - V$NLS_PARAMETERS

SQL 功能

OceanBase 支持 Oracle 中核心的 SQL 功能：

- 计划缓存
- 大纲绑定
- 计划管理演进
- 代价优化器
- 代价改写
- 预编译语句
- 全局索引
- 函数索引

外键

- OceanBase 支持外键。
- OceanBase 不支持添加外键约束 DISABLE 和 ENABLE。
- OceanBase 不支持 ALTER TABLR 语句中添加外键约束。
- OceanBase 不支持级联中的 SET NULL。

触发器

- OceanBase 目前仅支持行级触发器。
- OceanBase 目前只支持在表上创建触发器，不支持在视图上创建触发器。
- OceanBase 不支持对触发器使用 DISABLE 和 ENABLE 操作。

数据库链接

- OceanBase 暂不支持异构数据库的链接。

- OceanBase 暂不支持数据库链接的写语句。
- OceanBase 暂不支持部分子查询（主要是不能提升的子查询）。
- OceanBase 暂不支持 LINK 算子的 RESCAN 操作（执行时报错）。
- 更多信息请参阅文档 引用远程数据库中的对象 。

同义词

OceanBase 支持对表、视图、同义词和序列等对象创建同义词，并且支持创建公共同义词。

可更新视图

OceanBase 不支持 WITH CHECK OPTION 子句。

约束

- OceanBase 支持 CHECK、UNIQUE 和 NOT NULL 约束。
- OceanBase 不支持 UNIQUE 约束的 DISABLE 操作。

Hint

Oracle 中有 Hint 73 个，目前 OceanBase 兼容 25 个。另外，OceanBase 特有 Hint 20 个：

OceanBase 兼容 Oracle 的 Hint	OceanBase 特有的 Hint
USE_BNL	INDEX Hint
NO_USE_BNL Hint	FULL Hint
USE_PX Hint	LEADING Hint
NO_USE_PX Hint	ORDERED Hint
USE_JIT Hint	USE_MERGE Hint
NO_USE_JIT Hint	NO_USE_MERGE Hint
USE_HASH_AGGREGATION Hint	USE_HASH Hint
NO_USE_HASH_AGGREGATION Hint	NO_USE_HASH Hint
USE_LATE_MATERIALIZATION Hint	USE_NL Hint
NO_USE_LATE_MATERIALIZATION Hint	NO_USE_NL Hint
USE_NL_MATERIALIZATION Hint	PARALLEL Hint
NO_USE_NL_MATERIALIZATION Hint	PQ_DISTRIBUTE Hint
PLACE_GROUP_BY Hint	NO_REWRITE Hint
NO_PLACE_GROUP_BY Hint	NO_EXPAND Hint
NO_PRED_DEDUCE Hint	USE_CONCAT Hint
READ_CONSISTENCY Hint	MERGE Hint
FROZEN_VERSION Hint	NO_MERGE Hint
QUERY_TIMEOUT Hint	UNNEST Hint
LOG_LEVEL Hint	NO_UNNEST Hint

USE_PLAN_CACHE Hint	QB_NAME Hint
TRANS_PARAM Hint	
TRACING Hint	
STAT Hint TOPK Hint	
TRACE_LOG Hint	

安全相关

透明加密

OceanBase 支持 Oracle 兼容的 TDE 功能， 重做日志文件（ Redo Log ）加密暂不支持。

审计

- OceanBase 支持 Oracle 的标准审计，但是暂不支持统一审计。
- OceanBase 审计结果支持存放于文件或者内部审计表中。
- OceanBase 支持语句和对象审计类型，对象审计只支持表、序列和包对象, 其他对象暂不支持。
- OceanBase 暂不支持网络审计和 FGA 细粒度审计。

标签安全

- OceanBase 支持 Oracle 中的标签安全（Label Security）功能。
- OceanBase 暂不支持策略在创建生效后的 DISABLE 操作以及 DISABLE 后的 ENABLE 操作。

传输链路加密SSL

OceanBase 支持客户端与 OceanBase 服务器的传输链路加密。

分区支持

OceanBase 支持一级，二级分区，支持哈希（Hash）、范围（Range）和列表（List）等分区形式。

二级分区支持如下表：

分区/子分区	哈希	范围	列表
哈希	不支持	支持	支持
范围	支持	不支持	支持
列表	支持	支持	不支持

- OceanBase 中分区维护操作命令只支持基本的，例如一级分区 ADD PARTITION 操作。
- OceanBase 中暂不支持复杂分区维护操作，例如 SPLIT、MERGE 和 EXCHANGE 分区等。
- OceanBase 中不支持 TRUNCATE 分区。
- OceanBase 中二级分区只支持同构形态的，不支持异构形态的二级分区。

并行查询

- OceanBase 支持类 Oracle 的并行查询，OceanBase 中 DOP 需要手工指定，且暂不支持 Auto DOP 功能。
- OceanBase 暂不支持 PDML。

2 SQL 概述

结构化查询语言（Structured Query Language）简称 SQL，是一种有特殊目的的编程语言。和当下流行的其他关系数据库一样，所有程序和用户都可以使用 SQL 来访问 OceanBase 数据库中的数据。即便有一些平台、工具允许用户直接通过接口或界面的方式访问数据库，但这些平台、工具底层实际上依旧是使用 SQL 来访问数据库。

SQL 的历史

1970 年 6 月，IBM 公司 San Jose，California 实验室的 E. F. Codd 博士在 ACM（Association for Computing Machinery）期刊上发表了论文《大型共享数据银行的关系模型》（A Relational Model of Data for Large Shared Data Banks）并首次提出了关系模型的概念。

1974 年，同实验室的 D.D.Chamberlin 和 R.F.Boyce 在 IBM 公司研制的关系数据库系统 SystemR 中，研制出了一套规范语言 SEQUEL（Structured English QUEry Language），并在 1976 年 11 月的 IBM Journal of R&D 上公布了新版本的 SQL（称为 SEQUEL/2，1980年改名为 SQL）。

1979 年，Oracle 公司首先提供商用的 SQL，同时 IBM 公司在 DB2 和 SQL/DS 数据库系统中也实现了 SQL。

时至今日，SQL 已经成为了关系数据库管理系统（Relational Database Management System：RDBMS）的标准语言。

SQL 的标准

1986 年 10 月，美国国家标准协会 ANSI 采用 SQL 作为关系数据库管理系统的标准语言，并命名为 ANSI X3.135-1986，后来国际标准化组织（ISO）也采纳 SQL 作为国际标准。

1989 年，ANSI 采纳并使用了在 ANSI X3.135-1989 报告中定义的 SQL 标准语言，并称之为 ANSI SQL 89，该标准替代了之前的 ANSI X3.135-1986 版本。

下面是 SQL 发展的简要历史：

- 1986年，ANSI X3.135-1986，ISO/IEC 9075:1986，SQL-86。
- 1989年，ANSI X3.135-1989，ISO/IEC 9075:1989，SQL-89。
- 1992年，ANSI X3.135-1992，ISO/IEC 9075:1992，SQL-92（SQL2）。
- 1999年，ISO/IEC 9075:1999，SQL:1999（SQL3）。
- 2003年，ISO/IEC 9075:2003，SQL:2003。
- 2008年，ISO/IEC 9075:2008，SQL:2008。
- 2011年，ISO/IEC 9075:2011，SQL:2011。

现在，绝大多数被提及的 SQL 标准，其中涉及的内容其实都是 SQL 92 里最基本、最核心的一部分。OceanBase 目前也遵循的是 SQL 92 标准。

SQL 的运行

SQL 是用来访问关系数据库，如 OceanBase、Oracle 和 MySQL 的接口，所有的 SQL 语句都是对数据库的指令。

通常，SQL 可以分为 5 个部分：

1. 数据查询语言 DQL（Data Query Language）：也称为数据检索语言，用以从表中获得数据，并描述怎样将数据返回给程序输出。DQL 并不改变数据库中存储的数据内容。
2. 数据操作语言 DML（Data Manipulation Language）：用以改变数据库中存储的数据内容，即增加、修改和删除数据。
3. 事务控制语言 TCL（Transaction Control Language）：保证数据库的完整性、一致性，在同一个事务中的 DML 语句要么同时成功，要么同时失败。
4. 数据控制语句 DCL（Data Control Language）：对数据访问权限控制的命令。可以控制特定账号对特定数据库资源的访问权限。
5. 数据定义语言 DDL（Data Definition Language）：对数据库中资源进行定义、修改和删除，如新建表和删除表等。

SQL 的移植性

SQL 是访问数据库的标准语言，所有的主要关系数据库都支持 SQL，因此所有用 SQL 编写的程序都是可移植的。通常进行少量的修改就可以从一个关系数据库移植到另一个关系数据库上。

词汇惯例

- **粗体** 表示与操作或以文本或词汇表定义的术语相关联的图形用户界面元素。
- 保留字、关键字、标识符和参数中的大小写不敏感。为方便阅读与识别，这些字会以大写形式书写。
- 在不同的编程环境中，SQL 语句终止方式不同。本文档中以分号“;”来标识一个 SQL 的结尾。
- `行内代码` 表示文档中引用的代码。
- 为了突出重要信息，本文档会加粗“说明”、“注意”和“重要”等文字。
- 本文档中可选参数文本用方括号括起，如 [-n, -quiet]。

3 伪列

伪列（Pseudocolumn）的行为与表中的列相同，但并未存储具体数值。因此，伪列只具备读属性，您不可以对伪列进行插入、更新、删除的等行为。

注意：OceanBase 不支持 ROWID 伪列。

层次化查询伪列

层次化查询伪列仅在层次化查询中有效，要在查询中定义层次结构关系，必须使用 CONNECT BY 子句。层次化查询伪列有：

- CONNECT_BY_ISCYCLE
- CONNECT_BY_ISLEAF
- LEVEL

CONNECT_BY_ISCYCLE 伪列

CONNECT_BY_ISCYCLE 伪列用来协助标记循环是从哪一行开始的。如当前行的子节点同时也是其祖先节点之一，则CONNECT_BY_ISCYCLE 返回 1，否则返回 0。

CONNECT_BY_ISCYCLE 需要配合 CONNECT BY 子句的 NOCYCLE 使用，否则查询结果会因结构化树状节点构成环所导致的循环而报错。

CONNECT_BY_ISLEAF 伪列

CONNECT_BY_ISLEAF 伪列用来协助标记层次结构的叶子节点。如当前行无子节点并且为树的叶子节点时，返回 1，否则返回 0。

LEVEL 伪列

LEVEL 伪列用来协助标记节点的层次。层次结构中，根为第 1 层，根的子结点为第 2 层，之后以此类推。例如，根节点的 LEVEL 值会返回 1，根节点的子节点的 LEVEL 值会返回 2，之后以此类推。

序列伪列

序列（Sequence）伪列是数据库按照一定规则生成的自增数字序列。因其自增的特性，通常被用作主键和唯一键。序列伪列有两种取值方法：

- CURRVAL：返回序列的当前值。
- NEXTVAL：返回序列的下一个自增值。

使用序列伪列时，必须在 CURRVAL、NEXTVAL 前带上序列的名称，并用句点（.）引用。例如，序列的名称为 **SEQ_FOO**，则可以通过 SEQ_FOO.CURRVAL 获取 **SEQ_FOO** 序列的当前值。同样，可以通过 SEQ_FOO.NEXTVAL 获取 **SEQ_FOO** 序列的下一个自增值。

序列伪列的应用场景

序列伪列 CURRVAL 和 NEXTVAL 的值可以用于以下位置：

- 非子查询、物化视图或者视图中的 SELECT 语句的选择列表中。
- INSERT 语句中子查询的选择列表中。
- INSERT 语句中的 VALUE 子句中。
- UPDATE 语句中的 SET 子句中。

序列伪列 CURRVAL 和 NEXTVAL 的值不能用于以下位置：

- DELETE、SELECT 或者 UPDATE 语句的子查询中。

- 视图或者物化视图的查询中。
- 带 DISTINCT 运算符的 SELECT 语句中。
- 带 GROUP BY 子句或者 ORDER BY 子句的 SELECT 语句中。
- 与另一个 SELECT 语句通过 UNION、INTERSECT 或者 MINUS 集合运算符进行联合的 SELECT 语句中。
- SELECT 语句的 WHERE 子句中。
- CREATE TABLE 或者 ALTER TABLE 语句中列的 DEFAULT 值。
- CHECK 约束的条件中。

如何使用序列伪列

创建序列时，需要明确其初始值和步长。第一次引用 NEXTVAL 将返回序列的初始值，后续对 NEXTVAL 的引用将按照上一次序列的返回值加上序列定义的步长后返回一个新值。任何时候对 CURRVAL 的引用，都将返回当前序列的值，即最后一次对 NEXTVAL 引用时返回的值。

在会话中引用序列的 CURRVAL 伪列前，都应首先应用序列的 NEXTVAL 伪列来初始化本次会话的序列值。

创建序列时，可以定义其初始值以及其值之间的增量。对 NEXTVAL 的第一次引用将返回序列的初始值。对 NEXTVAL 的后续引用将会使序列值按照定义的增量递增，并返回新值。任何对 CURRVAL 的引用总是返回该序列的当前值，即最后一次对 NEXTVAL 引用时返回的值。对序列的创建的相关内容，请参考文档 CREATE SEQUENCE 章节。

在单条 SQL 语句中引用 NEXTVAL 时，OceanBase 按照以下方式递增序列：

- SELECT 语句的外部查询块每返回一行递增一次。这类查询块可以出现在以下地方：
  - 顶层 SELECT 语句。
  - INSERT... SELECT 语句。对于多表插入，NEXTVAL 必须位于 VALUES 子句中，子查询每返回一行序列就递增一次，即使多个分支引用了 NEXTVAL。
  - CREATE TABLE ... AS SELECT 语句。
  - CREATE MATERIALIZED VIEW ... AS SELECT 语句。
- UPDATE 语句每更新一行序列就递增一次。
- 每有一条包含 VALUES 子句的 INSERT 语句就递增一次。
- MERGE 语句每合并一行序列递增一次。NEXTVAL 可以出现在 merge_insert_clause 或者 merge_update_clause 子句中，也可两者同时出现。NEXTVAL 会随着每一行的更新和插入而递增，即使序列数值没有用于更新或者插入操作。如果 NEXTVAL 在这些位置中被指定了多次，那么对应每一行都递增一次序列，而且该行中出现的所有 NEXTVAL 都返回相同的值。

当这些位置多次引用一个序列的 NEXTVAL 伪列时，该序列都只递增一次，即为所有被引用的 NEXTVAL 伪列返回当前序列的下一个序列值。

当这些位置同时引用一个序列的 CURRVAL 和 NEXTVAL 伪列时，OceanBase 将递增该序列，即为被引用的 CURRVAL 和 NEXTVAL 伪列都返回当前序列的下一个序列值。

序列可以同时被许多用户访问，不存在等待和锁定。

ROWSCN 伪列

ORA_ROWSCN 伪列将最新更改的系统更改号（SCN：System Change Number）反映到一行。这个更改行为有粗粒度的块级（Block）和细粒度的行级（Row），这取决于 ROWDEPENDENCIES 字段。

ROWNUM 伪列

ROWNUM 伪列会对查询结果中的每一行进行编号，其值为该行在查询结果集中的具体位置。第一行返回值 1，第二行返回值 2，之后以此类推。

ROWNUM 的用法说明

ROWNUM 可以限制返回的行数，例如以下示例，将返回 **employees** 表中的 5 条数据：

```
SELECT * FROM employees WHERE rownum <= 5;
```

如果在 ROWNUM 后有 ORDER BY 子句，则将对满足 WHERE 条件句的结果进行重排序。如果将 ORDER BY 子句嵌入子查询中，并将 ROWNUM 伪列作为条件放置在顶级查询中，则可以强制 ROWNUM 条件在行排序之后执行。例如，使用以下语句查询年龄最大的 5 名员工信息是得不到预期结果的，该语句只是将查询结果中的前 5 条员工信息进行年龄排序：

```
SELECT * FROM employees WHERE rownum <= 5 ORDER BY age DESC;
```

正确的用法应该如下：

```
SELECT * FROM (SELECT * FROM employees ORDER BY age DESC) WHERE rownum <= 5;
```

在 WHERE 子句中，指定 ROWNUM 大于任何一个正整数时总是返回 **FALSE**，例如以下 SQL 语句将不返回任何信息：

```
SELECT * FROM employees WHERE rownum > 1;
```

因为在获得表的第一行结果时，改行的 ROWNUM 伪列值将被赋值为 1，此时在 WHERE 条件判断时结果为 **FALSE**，则此行被舍去。在获得第二行结果时，该行的 ROWNUM 伪列值任然被赋值为 1，WHERE 条件判断的结果依旧为 **FALSE**，此行再次被舍去。以此类推，所以所有行都不满足条件，因此不返回任何数据。

也可以通过 UPDATE 语句将ROWNUM 数值赋值给表中的某一列，例如：

```
UPDATE employees SET id = rownum;
```

此语句将对表 **employees** 的 **id** 列进行 ROWNUM 赋值，即依次对 **id** 列赋值 1、2、... 直至该表总行数。

注意：在查询中使用 ROWNUM 可能会影响视图优化。

4 基本元素

4.1 内建数据类型

4.1.1 内建数据类型概述

OceanBase 数据库操作的每个值都有一个数据类型。值的数据类型将一组固定的属性与值相关联，这些属性使 OceanBase 将一种数据类型的值与另一种数据类型的值区别对待。OceanBase 数据库提供了许多内建数据类型， 这些内建数据类型也称 OceanBase 基本数据类型。

OceanBase 支持如下数据类型，与 Oracle 数据类型保持一致：

- 字符数据类型
- 数值数据类型
- 日期时间数据类型
- RAW 数据类型
- 大对象数据类型

字符数据类型

字符数据类型在数据库字符集或国家字符集中存储字符（字母数字）数据，即单词和自由格式的文本。字符数据类型与其他数据类型相比具有限制性，因此属性较少。

字符数据存储在字符串中，其字节值与创建数据库时指定的字符集之一相对应。OceanBase 支持单字节和多字节字符集。

说明：字符数据类型的列可以存储所有字母数字值，但是 NUMBER 数据类型的列只能存储数字值。

数据类型	长度	使用说明	长度说明
CHAR(size [BYTE | CHAR])	定长	索引效率高，程序里面使用 trim 去除多余的空白。	参数 size 必须是一个介于 1~2000 之间的数值，存储大小为 size 个字节。
NCHAR[(size)]	定长	使用 UNICODE 字符集（所有的字符使用两个字节表示）。	参数 size 必须是一个介于 1~2000 之间的数值，存储大小为 size 字节的两倍。
NVARCHAR2(size)	变长	使用 UNICODE 字符集（所有的字符使用两个字节表示）。	参数 size 的值必须介于 1~32767 之间，字节的存储大小是所输入字符个数的两倍。
VARCHAR2(size [BYTE | CHAR])	变长	使用 UNICODE 字符集（所有的字符使用两个字节表示）。	参数 size 必须是一个介于 1~32767 之间的数值，存储大小为输入数据的字节的实际长度，而不是 size 个字节。
VARCHAR(size [BYTE | CHAR])	变长	OceanBase 中 VARCHAR 和 VARCHAR2 没有区别。	参数 size 必须是一个介于 1~32767 之间的数值，存储大小为输入数据的字节的实际长度，而不是 size 个字节。

其中，CHAR 和 VARCHAR2 数据类型需要指定 length 语义，其默认值由系统变量 NLS_LENGTH_SEMANTICS 控制。

数值数据类型

OceanBase 为我们提供了四种存储数值的数据类型，它们分别是 NUMBER、FLOAT、BINARY_FLOAT 和

BINARY_DOUBLE。您可以通过这四种数值类型存储定点数、浮点数和零。在数值计算时，数值类型具有不同的优先级，具体信息请查阅 数值类型的优先级 。

数据类	长度（字节）	说明
NUMBER	4~40	NUMBER(p,s) 存储变长、十进制精度的定点数；也可以存储浮点数，此时 NUMBER 没有 p 和 s。
FLOAT	4~40	FLOAT(p) 是 NUMBER 数据类型的子类型。二进制精度范围为 1~126。FLOAT 不是浮点数。
BINARY_FLOAT	4	二进制精度浮点数，是一种 32 位单精度浮点数数据类型。
BINARY_DOUBLE	8	二进制精度浮点数，是一种 64 位双精度浮点数数据类型。

日期时间数据类型

与 Oracle 中的数据类型一致，OceanBase 也支持日期时间和间隔数据类型。日期时间数据类型被用来在数据库中保存日期和时间信息。与日期时间数据类型存储特定的一个时间点不同，间隔数据类型用来存储一段时间，可以有效表示了两个日期时间值之间的差异。

- 日期时间数据类型：DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE。
- 间隔数据类型：INTERVAL YEAR TO MONTH 和 INTERVAL DAY TO SECOND。

类型	格式	取值范围	说明
DATE	DD-MON-RR	0001-01-01 00:00:00 ~ 9999-12-31 23:59:59	存储日期和时间信息，精确到秒，不带时区。
TIMESTAMP [(scale)]	DD-MON-RR HH.MI.SSXFF AM	0001-01-01 00:00:00.000000000 ~ 9999-12-31 23:59:59.999999999	DATE 数据类型的扩展，精确到纳秒，不带时区。
TIMESTAMP [(scale)] WITH TIME ZONE	DD-MON-RR HH.MI.SSXFF AM TZR	0001-01-01 00:00:00.000000000 ~ 9999-12-31 23:59:59.999999999	DATE 数据类型的扩展，精确到纳秒，带时区信息。
TIMESTAMP [(scale)] WITH LOCAL TIME ZONE]	DD-MON-RR HH.MI.SSXFF AM	0001-01-01 00:00:00.000000000 ~ 9999-12-31 23:59:59.999999999	带有本地时区的 TIMESTAMP 值，存储数据库时区。
INTERVAL YEAR TO MONTH	INTERVAL ‘year-month’ YEAR(percision) TO MONTH	由参数 percision 确定 YEAR 元素的精度。	存储以年和月为单位的时间段。
INTERVAL DAY TO SECOND	INTERVAL ‘dd hh:mm:ss’ DAY(precision) TO	由参数 precision 和 fractional_seconds_precision 确定 DAY 元素	存储以天、小时、分钟和秒为单位的时间段。

	SECOND(fractional_seconds_precision)	的精度和 SECOND 元素小数部分的精度。	

TIMESTAMP [(scale)]、TIMESTAMP [(scale)] WITH TIME ZONE、TIMESTAMP [(scale)] WITH LOCAL TIME ZONE 的值域和精度取决于 scale。该属性表示小数部分的最大有效位数，取值范围 0~9，默认值为 6。

其中，时间类型的格式受系统变量 NLS_DATE_FORMAT 、 NLS_TIMESTAMP_FORMAT 和 NLS_TIMESTAMP_TZ_FORMAT 控制。

日期时间和间隔可以进行计算，具体信息请查阅 日期时间 DATE 和间隔 INTERVAL 的计算 。

RAW 数据类型

OceanBase 中用于保存二进制数据或字节字符串的数据类型是 RAW ，是一种可变长度的数据类型，在不同平台上传输时，传送的都是二进制信息，即使字符集不同也不需要转换。

大对象数据类型

大对象数据类型（Large Object），简称 LOB。用来存储大型和非结构化数据，例如文本、图像、视频和空间数据等。

类型	长度	定义长度上限（字节）	字符集
BLOB	变长	48M	BINARY
CLOB	变长	48M	UTF8MB4

4.1.2 CHAR 数据类型

CHAR 数据类型指定了固定长度的字符串。OceanBase 确保存储在 CHAR 数据类型列中的所有值的长度等于 size 指定的固定长度。如果您插入的值短于指定长度，则 OceanBase 将用空格填充剩余长度。如果您插入的值超过列长度，则 OceanBase 会返回错误。

BYTE 和 CHAR 限定符会覆盖由 NLS_LENGTH_SEMANTICS 参数指定的语义，而该参数默认字节语义。为了确保具有不同字符集的数据库之间，能够进行正确的数据转换，CHAR 数据必须由格式正确的字符串组成。

语法

```
CHAR [(size [BYTE | CHAR])]
```

参数

参数	说明
size	表示指定的固定长度
BYTE	表示以字节（BYTE）为单位提供列的长度
CHAR	表示以字符（CHAR）为单位提供列的长度

CHAR 数据类型的列的默认长度为 1 个字节，允许的最大长度为 2000 个字节。

示例

当创建带有 CHAR 数据类型列的表时，默认情况下，您以字节为单位提供列的长度。BYTE 与默认值相同。如果使用 CHAR 限定符。则以字符为单位提供列长。

示例 1

将一个 1 字节的字符串插入 CHAR(10) 类型的列中，在存储该字符串之前，将为其空白填充为 10 个字节。

示例 2

CHAR(10 CHAR)，表示以字符为单位提供列长。

4.1.3 NCHAR 数据类型

NCHAR 是 UNICODE 字符数据类型，长度固定。您在创建数据库时，列的最大长度由国家字符集定义确定。创建包含 NCHAR 数据类型列的表时，以字符定义列长度。字符数据类型 NCHAR 的宽度规范是指字符数。允许的最大列大小为 2000 字节。

如果希望用较少的空间来存储中文，可以选择 NCHAR 类型。

使用 NCHAR 来存储数据时，如果存储的数据没有达到指定长度，那么数据库将自动补足空格。指定长度时，采用 CHAR 为计量单位，不可以手动指定其他单位。

注意：您不能将 CHAR 值插入 NCHAR 列，也不能将 NCHAR 值插入 CHAR 列。

语法

```
NCHAR[(size)]
```

参数

参数	说明
size	表示定长字符数据的长度大小，最大大小由国家字符集定义确定，上限为 2000 个字节。默认定长字符数据的长度最小为 1 个字符。

更多信息

UNICODE 字符集

UNICODE 字符集是对字符的一种编码，具体的编码方式有 UTF-8、UTF-16、UTF-32 和压缩转换等，编码方式决定了一个字符的存储大小，中英文在不同的存储方式上使用的空间是不一样的。

三种编码方式的对比

编码方式	编码字节数	BOM	优点	缺点

UTF-8	不定长编码方式，单字节（ASCII 字符）或多字节（非 ASCII 字符）；最小 Code Unit 是 8 位。	无字节序：如果一个文本的开头有字节流 EF BB BF，表示是 UTF-8 编码。	较为理想的 UNICODE 编码方式：与 ASCII 编码完全兼容；无字节序；自同步和纠错能力强，适合网络传输和通信；扩展性好。	变长编码方式不利于程序内部处理。
UTF-16	双字节或者四字节；最小 Code Unit 是 16 位。	有字节序：UTF-16LE（小端序）以 FF FE 代表，UTF-16BE（大端序）以 FE FF 代表。	最早的 Unicode 编码方式，已被应用于大量环境中；适合内存中 Unicode 处理；很多编程语言的 API 中作为 string 类型的编码方式。	无法兼容于 ASCII 编码；增补平面码点编码使用代理对，编码复杂；扩展性差。
UTF-32	固定四字节；最小 Code Unit 是 16 位。	有字节序：UTF-16LE（小端序）以 FF FE 代表，UTF-16BE（大端序）以 FE FF 代表。	固定字节编码读取简单，编译程序内部处理；Unicode 码点和 Code Unit 一一对应关系。	所有字符以固定四字节编码，浪费存储空间和带宽；与 ASCII 编码不兼容；扩展性差；实际使用少。

数据库字符集

- 用来存储 CHAR、VARCHAR2、CLOB 等类型数据。
- 用来标示诸如表名、列名以及 PL/SQL 变量等。
- 用来存储 SQL 和 PL/SQL 程序单元等。

国家字符集

- 用以存储 NCHAR、NVARCHAR2、NCLOB 等类型数据。
- 国家字符集实质上是为 OceanBase 选择的附加字符集，主要作用是为了增强 OceanBase 的字符处理能力，NCHAR 数据类型使用国家字符集，在使用 CHAR 数据类型提供的数据库字符集的同时，NCHAR 类型提供了除了数据库字符集之外的另一种字符集选择。

4.1.4 NVARCHAR2 数据类型

NVARCHAR2 是 UNICODE 字符数据类型。列的长度可变，最大存储长度为 32767 个字节，最小值为 1 个字节。创建包含 NVARCHAR2 数据类型列的表时，您必须为 NVARCHAR2 指定可以容纳的最大字符数，默认采用 CHAR 为计量单位，不可以手动指定其他单位。

如果不确定存储的数据长度，而且有可能包含中文，可以选择 NVARCHAR2 类型。

格式

```
NVARCHAR2(size)
```

参数

参数	说明
size	表示列的长度，可变长度。您必须为 NVARCHAR2 指定大小，对于 AL16UTF16 编码，字节数最大为两倍，对于 UTF8 编码，字节数最大为三倍。字节数大小由国家字符集定义确定，上限为 32767 个字节。

4.1.5 VARCHAR2 数据类型

VARCHAR2 数据类型存储一个可变长度的字符串，最大长度为 32767。当您创建 VARCHAR2 列时，您必须为 VARCHAR2 列指定最大长度。尽管允许存储的实际字符串为零长度字符串（’ ’），但该最大值必须至少为 1 个字节。OceanBase 会将每个值完全按照您指定的方式存储在列中。

注意：

- 从技术上讲，字符是数据库字符集的代码点。
- 创建具有该列或属性，且在列或属性定义中未包含任何明确限定词的数据库对象时，长度语义由会话的 NLS_LENGTH_SEMANTICS 参数的值决定。
- OceanBase 使用未填充的比较语义比较 VARCHAR2 值。
- 具有不同字符集的数据库之间进行正确的数据转换时，必须确保 VARCHAR2 数据由格式正确的字符串组成。

语法

```
VARCHAR2(size [BYTE | CHAR])
```

参数

参数	说明
size	表示存储的字节数或字符数的长度大小。
BYTE	表示该列将具有字节长度的语义。
CHAR	指示该列将具有字符语义。

示例

您可以使用 CHAR 限定符，以字符（而不是字节）为单位提供最大长度。

```
VARCHAR2(10 CHAR)
```

您可以使用 BYTE 限定符，以字节为单位明确给出最大长度。

```
VARCHAR2(10 BYTE)
```

4.1.6 VARCHAR 数据类型

VARCHAR 数据类型用于存储可变长度的字符串。当您创建 VARCHAR 数据类型的列时，您必须为 VARCHAR 列指定最大长度。尽管允许存储的实际字符串为零长度字符串（’ ’），但该最大值必须至少为 1 个字节。OceanBase 会将每个值完全按照您指定的方式存储在列中。如果插入值超过指定长度，OceanBase 会返回错误。

在 OceanBase 中 VARCHAR 数据类型和 VARCHAR2 数据类型没有区别，一般会用 VARCHAR2。

语法

```
VARCHAR(size [BYTE | CHAR])
```

参数

参数	说明
size	表示存储的字节数或字符数的长度大小。
BYTE	表示该列将具有字节长度的语义。
CHAR	指示该列将具有字符语义。

4.1.7 NUMBER 数据类型

NUMBER 是变长、精确的数值类型，占 4~40 字节存储空间，其中 4 字节存放 NUMBER 的元数据信息，0~36 字节存放 NUMBER 的具体数值。可以存储零、浮点数、正的定点数和负的定点数，其绝对值范围 1.0 × 10-130 ~ 1.0 × 10126（不包括 1.0 × 10126）。如果您指定的算术表达式的绝对值大于或等于 1.0 × 10-130 ，OceanBase 会返回错误。

NUMBER 数据类型具有较好的数据精确度、通用性强、可移植性强，运算效率相对浮点类型偏低。

语法

```
NUMBER [(p[s])]
```

参数

参数	取值范围	说明
p	1~38	表示精度，最大有效十进制数字，其中最高有效数字是最左边的非零数字，最低有效数字是最右边的已知数字。
s	-84~127	表示小数位数，从小数点到最低有效数字的位数。比例尺范围是 -84~127。

说明

- 若 s>0，精确到小数点右边 s 位，四舍五入。然后检验有效位是否小于等于 p。
- 若 s<0，精确到小数点左边 s 位，四舍五入。然后检验有效位是否小于等于 p + |s|。
- 若 s=0， 表示整数。

注意：

- 小数位数正标度是小数点右边到最低有效位数（包括最低有效位数）的有效位数。精度和小数位都用十进制数字表示。
- 小数位数负数标度是小数点左边的有效位数，但不包括最低有效位数。对于负比例，最低有效数字在

小数点的左侧，因为实际数据四舍五入到小数点左侧的指定位数。

示例

示例 1： 使用以下格式指定整数。

NUMBER(p) 表示精度为 p 且标度为 0 的定点数，等效于 NUMBER(p,0)。NUMBER 表示浮点数时，缺少精度和小数位数指示符。

示例 2：使用不同的精度和小数位数存储数据。为了避免 OceanBase 存储的数据超过精度，我们需要为定点数字列指定小数位数和精度，对输入进行额外的完整性检查。但这不会强制固定定点数字列的长度。若实际存储的数据超过精度，则 OceanBase 会返回错误。如果存储的数据超过小数位数，则 OceanBase 对其进行四舍五入。

实际数据	指定为	存储为
123.89	NUMBER	123.89
123.89	NUMBER(3)	124
123.89	NUMBER(3,2)	超过精度
123.89	NUMBER(4,2)	超过精度
123.89	NUMBER(5,2)	123.89
123.89	NUMBER(6,1)	123.9
123.89	NUMBER(6,-2)	100
.01234	NUMBER(4,5)	.01234
.00012	NUMBER(4,5)	.00012
.000127	NUMBER(4,5)	.00013
.000012	NUMBER(2,7)	.000012
.0000123	NUMBER(2,7)	.000012
1.2e-4	NUMBER(2,5)	.00012
1.2e-5	NUMBER(2,5)	.00001

4.1.8 FLOAT 数据类型

FLOAT 数据类型是具有精度（precision）的 NUMBER 数据类型的子类型，需要占 4~40 字节存储空间。它的精度是按二进制有效位数计算的，范围为 1~126，小数位数不可自定义。FLOAT 类型为变长、非精确数值类型。

语法

```
FLOAT [(p)]
```

参数

参数	定义	范围	说明
p	精度	1~126	定义数值精度，按二进制有效位数计算，转换为十进制精度要乘以 0.30103。

说明：

- 二进制精度转换为十进制精度的换算关系： 二进制精度 = int(十进制精度 x 0.30103)。
- 十进制精度转换为二进制精度的换算关系： 十进制精度 = int(二进制精度 x 3.32193)。

示例

示例 1：使用 FLOAT 设置二进制精度为 2，转换为十进制精度 int(2 x 0.30103) = 0.6，结果向下取整，则 FLOAT(2)的十进制精度为 0。

```
FLOAT(2)
```

示例 2：创建 **test** 表，并向里面插入数据。其中 **col1** 列是 NUMBER 类型，**col2** 列是 FLOAT 类型。NUMBER(5,2)表示十进制精度的定点数，有效位数为 5，结果保留小数点后 2 位。FLOAT(5) 的二进制精度为 5，转换为十进制精度为int(5 x 0.30103) = 1.50515，向下取整后十进制精度为 1。如 **col2** 列的 123.45 用科学计数法表示为 1.2345 x 102，1.2345 小数点后保留 1 位，四舍五入变为 1.2，查询结果显示为 1.2 x 10 2 = 120。执行以下语句：

```
CREATE TABLE test (col1 NUMBER(5,2), col2 FLOAT(5));
INSERT INTO test VALUES (1.23, 1.23);
INSERT INTO test VALUES (7.89, 7.89);
INSERT INTO test VALUES (12.79, 12.79);
INSERT INTO test VALUES (123.45, 123.45);
```

执行以下语句查看 **test** 表：

```
SELECT * FROM test;
```

返回结果如下：

```
+-----------+----------+
| col1 | col2 |
+-----------+----------+
| 1.23 | 1.2 |
+-----------+----------+
| 7.89 | 7.9 |
+-----------+----------+
| 12.79 | 13 |
+-----------+----------+
| 123.45 | 120 |
+-----------+----------+
```

说明：当转换 ANSI FLOAT 数据时，您可以使用 OceanBase 数据库内部使用的 FLOAT 数据类型。但是 OceanBase 建议您改用 BINARY_FLOAT 和 BINARY_DOUBLE 浮点数字。

4.1.9 浮点数字

浮点数字（FlOAT-Point Numbers）可以有小数点，从第一位到最后一位，或者根本没有小数点。您可以选择

在数字后使用指数来增加范围，例如 1.666 e-20。小数位数不适用于浮点数，因为小数点后可以出现的位数不受限制。

注意：二进制浮点数与 NUMBER 的区别，在于值是由 OceanBase 数据库内部存储的。NUMBER 数据类型使用十进制精度存储，所存储的数据都会精确存储。二进制浮点数（FlOAT-Point Numbers）是使用二进制精度（数字 0 和 1）存储的，这种存储值的方式不能精确地表达使用十进制精度的所有值。

语法

OceanBase 数据库为浮点数提供了两种数字数据类型：

BINARY_FLOAT，是一种 32 位单精度浮点数数据类型。每个 BINARY_FLOAT 值需要 4 个字节。

BINARY_DOUBLE，是一种 64 位双精度浮点数数据类型。每个 BINARY_DOUBLE 值需要 8 个字节。

说明：

- 在 NUMBER 列中，浮点数具有十进制精度。
- 在 BINARY_FLOAT 或 BINARY_DOUBLE 列中，浮点数具有二进制精度。
- 二进制浮点数暂时不支持特殊值 infinity 和 NaN（不是数字）。

取值范围

您可以根据值域范围指定浮点数。

值	BINARY_FLOAT	BINARY_DOUBLE
最大正有限值	3.40282E+38F	1.79769313486231E+308
最小正有限值	1.17549E-38F	2.22507485850720E-308

更多信息

IEEE754 符合性

IEEE 标准 754-1985（IEEE754） OceanBase 浮点数据类型的实现基本上符合电气和电子工程师协会（IEEE）的二进制浮点算法标准。

浮点数据类型在以下方面符合 IEEE754

- SQL 函数 SQRT 实现平方根。
- SQL 函数 REMAINDER 实现余数。
- 算术运算符一致。
- 比较运算符符合要求。
- 转换运算符符合。
- 支持默认的舍入模式。
- 支持默认的异常处理模式。

- OceanBase 暂时不支持 INF、-INF、NaN、BINARY_FLOAT_NAN、BINARY_DOUBLE_NAN 之类的特殊常量。
- SQL 函数 ROUND 、TRUNC 、CEIL 和 FLOOR 提供了将二进制浮点数 BINARY_FLOAT 、BINARY_DOUBLE 舍入为整数值 BINARY_ FLOAT 和 BINARY_DOUBLE 的功能。
- SQL 函数 TO_CHAR、TO_NUMBER、TO_ NCHAR、TO_BINARY_FLOAT、TO_BINARY_DOUBLE 和 CAST 提供了将二进制浮点数 BINARY_FLOAT 、BINARY_DOUBLE 舍入为十进制数和十进制数舍入为 BINARY_FLOAT 、 BINARY_DOUBLE 的功能。

浮点数据类型在以下方面不符合 IEEE754：

- -0 被强制为 +0。
- 不支持与 NaN 比较。
- 不支持非默认的舍入模式。
- 不支持非默认异常处理模式。

4.1.10 数值类型的优先级

不同的数值数据类型在操作时有不同的优先级。在 OceanBase 中，BINARY_DOUBLE 的优先级最高，其次是 BINARY_FLOAT，最后是 NUMBER 。

在对多个数值进行操作时：

- 若有一个操作数为 BINARY_DOUBLE，OceanBase 会在执行该操作之前将所有操作数转换为 BINARY_DOUBLE。
- 若有一个操作数为 BINARY_FLOAT，OceanBase 会在执行该操作之前将所有操作数转换为 BINARY_FLOAT。
- 若有一个操作数都不是 BINARY_DOUBLE 和 BINARY_FLOAT，OceanBase 会在执行该操作之前将所有操作数转换为 NUMBER。
- 若所需要的转换失败，则该操作将失败。
- 与其他数据类型相比，数值数据类型的优先级低于日期时间与间隔数据类型，高于字符和所有其他数据类型。

4.1.11 DATE 数据类型

DATE 数据类型存储日期和时间信息。尽管日期和时间信息可以用字符和数字数据类型表示，但 DATE 数据类型具有特殊的关联属性。对于每个 DATE 值，OceaBase 存储以下信息：年、月、日、小时、分钟和秒，但是并不包含时区信息。

格式

DATE 数据类型的默认输入输出格式由 NLS_DATE_FORMAT 决定，运行以下 SQL 语句查看默认格式：

```
SELECT @@NLS_DATE_FORMAT FROM DUAL;
```

返回结果：

```
DD-MON-RR
```

如果您需要自定义数据的格式，可以使用转换函数。在插入数据时，您可以通过函数 TO_DATE (char,fmt) 指定数据的输入格式。查询数据时，您可以通过函数 TO_CHAR (datetime,fmt) 指定数据的输出格式。这两个转换函数会将字符串转化为参数 fmt 中定义的格式。在 fmt 未指明的情况下，使用默认格式。

注意： DATE 数据类型储存小时、分钟和秒等时间信息，但是默认格式中并不包含时间信息。

取值范围

0001-01-01 00:00:00~9999-12-31 23:59:59

示例

示例1： 返回当前的系统日期，由于未指定参数 fmt，所以 TO_CHAR 按数据类型的默认格式返回数据。

```
SELECT TO_CHAR(sysdate) FROM DUAL;
```

返回结果：

```
+------------------+
| TO_CHAR(SYSDATE) |
+------------------+
| 24-FEB-20 |
+------------------+
```

示例2： 在未指定 DATE 字面量的情况下，数据库将返回系统默认值：

- 年：当前年份，由 SYSDATE 返回。
- 月：当前月份，由 SYSDATE 返回。
- 日：01，默认是当前月的第一天。
- 小时、分钟和秒均为 0。

例如当前查询是 2020 年 2 月发出的：

```
SELECT TO_CHAR(TO_DATE('2020', 'YYYY'),'YYYY-MM-DD HH24:MI:SS') FROM DUAL;
```

返回结果：

```
+-----------------------------------------------------------+
| TO_CHAR(TO_DATE('2020', 'YYYY'),'YYYY-MM-DD HH24:MI:SS') |
+-----------------------------------------------------------+
| 2020-02-01 00:00:00 |
+-----------------------------------------------------------+
```

4.1.12 TIMESTAMP 数据类型

日期时间数据类型中除了 DATE 数据类型以外还有 TIMESTAMP[(scale)] 时间戳数据类型。它是 DATE 数据类型的扩展，跟 DATE 数据类型一样，它存储了年、月、日、时、分和秒等信息，但是不存储时区信息，它的时间最大可以精确到纳秒。所以它常用于存储时间精确度高和不需要考虑时区变换的数据。

语法

```
TIMESTAMP [(scale)]
```

参数

参数	取值范围	说明
scale	0~9	TIMESTAMP[(scale)] 的范围和精确度取决于 scale 的值，最大值为 9（纳秒，即秒数精确至小数点 9 位），最小值为 0（秒，即秒数精确至小数点 0 位），默认值为 6。

格式

TIMESTAMP 数据类型的默认输入输出格式由 NLS_TIMESTAMP_FORMAT 决定，运行以下 SQL 语句查看默认格式：

```
SELECT @@NLS_TIMESTAMP_FORMAT FROM DUAL;
```

返回结果：

```
DD-MON-RR HH.MI.SSXFF AM
```

如果您需要自定义数据的格式，可以使用转换函数。在插入数据时，您可以通过函数 TO_TIMESTAMP (char,fmt) 指定数据的输入格式。查询数据时，您可以通过函数 TO_CHAR (datetime,fmt) 指定数据的输出格式。这两个转换函数会将字符串转化为参数 fmt 中定义的格式。在 fmt 未指明的情况下，使用默认格式。

取值范围

0001-01-01 00:00:00.000000000~9999-12-31 23:59:59.999999999

示例

示例1：如下代码所示，在表 **Timestamp_Sample** 中创建了数据类型为 TIMESTAMP 的两列 **timestp1** 和 **timestp2**， 并且为 **timestp2** 指定了时间的精度为 3。向两列同时使用 TO_TIMESTAMP(string, format) 插入日期值 **2020-01-01 11:00:00**。

```
CREATE TABLE Timestamp_Sample(timestp1 TIMESTAMP, timestp2 TIMESTAMP(3));
INSERT INTO Timestamp_Sample(timestp1,timestp2) VALUES(TO_TIMESTAMP('2020-01-01 11:00:00','YYYY-MM-DD HH24:MI:SS'),TO_TIMESTAMP('2020-01-01 11:00:00','YYYY-MM-DD HH24:MI:SS'));
SELECT * FROM Timestamp_Sample;
```

结果如下，可以看到 **timestp1** 由于没有指定 scale 的值，所以结果时间精度默认为 6 位，**timestp2** 在指定后精度为 3 位：

```
+-----------------------------+--------------------------+
| timestamp1 | timestamp2 |
+-----------------------------+--------------------------+
| 01-JAN-20 11.00.00.000000 AM | 01-JAN-20 11.00.00.000 AM |
+-----------------------------+--------------------------+
```

示例2：以下语句使用函数 TO_CHAR (datetime,fmt) 指定输出格式。

```
SELECT TO_CHAR(TO_TIMESTAMP_TZ('25-FEB-20 11:00:00 AM America/Los_Angeles','DD-MON-RR HH:MI:SSXFF
PM TZR'),'YYYY-MM-DD HH:MI:SSXFF PM TZR') Timestamp
FROM DUAL;
```

返回结果：

```
+-------------------------------------------------+
| Timestamp |
+-------------------------------------------------+
| 2020-02-25 11:00:00.000000000 AM America/Los_Angeles |
+-------------------------------------------------+
```

4.1.13 TIMESTAMP WITH TIME ZONE 数据类型

TIMESTAMP [(scale)] WITH TIME ZONE 是 TIMASTAMP [(scale)] 的变体，与 TIMASTAMP [(scale)] 一样，它存储了年、月、日、时、分和秒等信息，阈值和精确度取决于 scale 的值。但是与 TIMASTAMP [(scale)] 不同的是它还可以存储时区信息，所以常用于存储跨地理区域的日期时间信息。

语法

```
TIMESTAMP [(scale)] WITH TIME ZONE
```

参数

参数	取值范围	说明
scale	0~9	TIMESTAMP [(scale)] WITH TIME ZONE 的阈值和精确度取决于 scale 的值，最大值为 9（纳秒，即秒数精确至小数点 9 位），最小值为 0（秒，即秒数精确至小数点 0 位），默认值为 6。

格式

TIMESTAMP WITH TIME ZONE 数据类型的默认输入输出格式由 NLS_TIMESTAMP_TZ_FORMAT 决定，运行以下 SQL 语句查看默认格式：

```
SELECT @@NLS_TIMESTAMP_TZ_FORMAT FROM DUAL;
```

返回结果：

```
DD-MON-RR HH.MI.SSXFF AM TZR
```

如果您需要自定义数据的格式，可以使用转换函数。在插入数据时，您可以通过函数 TO_TIMESTAMP_TZ (char,fmt) 指定数据的输入格式。查询数据时，您可以通过函数 TO_CHAR (datetime,fmt) 指定数据的输出格式。这两个转换函数会将字符串转化为参数 fmt 中定义的格式。在 fmt 未指明的情况下，使用默认格式。

取值范围

0001-01-01 00:00:00.000000000~9999-12-31 23:59:59.999999999

示例

以下示例都使用 TO_TIMESTAMP_TZ (char,fmt) 输入时间戳值。

在插入时区时， OceanBase 支持使用时区偏移量和时区区域名称：

- 时区偏移量：与格林尼治标准时间 GMT 的差（小时和分钟）
- 时区区域名称（TZR）及时区缩写（TZD）：国家/城市 时区缩写

使用时区偏移量

执行以下语句，使用时区偏移量插入数值：

```
SELECT TO_TIMESTAMP_TZ('2020-01-01 11:00:00 -05:00','YYYY-MM-DD HH:MI:SS TZH:TZM') FROM DUAL;
```

返回结果：

```
01-JUN-20 11.00.00.000000000 AM AMERICA/LOS_ANGELES
```

使用时区区域名称及时区缩写

执行以下语句，使用时区区域名称及缩写插入数值：

```
SELECT TO_TIMESTAMP_TZ('2020-01-01 11:00:00 America/Los_Angeles PST','YYYY-MM-DD HH:MI:SS TZR TZD')
FROM DUAL;
```

返回结果：

```
01-JUN-20 11.00.00.000000000 AM America/Los_Angeles PST
```

夏令时

OceanBase 支持夏令时且用时区缩写表示夏令时信息，以 America/Los_Angeles 为例，夏令时 PDT 从每年

的 3 月的第二个星期日到 11 月的第一个星期日执行，其余时间为 PST。当插入的值中只包含时区区域名称时，OceanBase 会根据插入的时间信息判断当前插入的时区区域是否在夏令时时段，并在返回的结果中会包含时区缩写，以此指明当前时间是夏令时。

您可以执行以下示例代码：

```
SELECT TO_TIMESTAMP_TZ('2020-02-01 11:00:00 America/Los_Angeles','YYYY-MM-DD HH:MI:SS TZR') FROM DUAL;
SELECT TO_TIMESTAMP_TZ('2020-06-01 11:00:00 America/Los_Angeles','YYYY-MM-DD HH:MI:SS TZR') FROM DUAL;
```

返回结果：

```
01-JUN-20 11.00.00.000000000 AM America/Los_Angeles PST
01-JUN-20 11.00.00.000000000 AM America/Los_Angeles PDT
```

4.1.14 TIMESTAMP WITH LOCAL TIME ZONE 数据类型

TIMESTAMP [(scale)] WITH LOCAL TIME ZONE 数据类型的时区信息是当前会话发生的时区。与 TIMASTAMP [(scale)] WITH TIME ZONE 的区别在于用户不需要自己输入时区信息，OceanBase 直接存储默认的数据库时区 +00:00 （不可更改）。当用户检索数据时，OceanBase 会返回本地会话时区（可更改）。此数据类型常用于始终在两层应用程序中的客户端系统的时区中显示的日期信息。

语法

```
TIMASTAMP [(scale)] WITH TIME ZONE
```

参数

参数	取值范围	说明
scale	0~9	TIMESTAMP [(scale)] 的阈值和精确度取决于 scale 的值，最大值为 9（纳秒），最小值为 0（秒），默认值为 6。

格式

TIMESTAMP WITH LOCAL TIME ZONE 数据类型的默认输入输出格式由 NLS_TIMESTAMP_FORMAT 决定，运行以下 SQL 语句查看日期时间格式：

```
SELECT @@NLS_TIMESTAMP_FORMAT FROM DUAL;
```

返回结果：

```
DD-MON-RR HH.MI.SSXFF AM
```

如果您需要自定义数据的格式，可以使用转换函数。您可以通过函数 TO_CHAR (datetime,fmt) 指定数据的输出格式。这个转换函数会将字符串转化为参数 fmt 中定义的格式。在 fmt 未指明的情况下 TO_CHAR 按数据类型

的默认格式返回数据。关于 TIMESTAMP WITH TIME ZONE 数据类型值的输入格式，您可以参阅文档 时间戳字面量 。

取值范围

0001-01-01 00:00:00.000000000~9999-12-31 23:59:59.999999999

示例

TIMESTAMP WITH TIME ZONE 数据类型没有专门的字面量为其赋值，时区信息由 **SESSIONTIMEZONE** 返回本地对话时区。**SESSIONTIMEZONE** 的值取自自定义参数 **TIME_ZONE**。

```
CREATE TABLE LocalTZ ( ltzcol TIMESTAMP WITH LOCAL TIME ZONE);
INSERT INTO LocalTZ VALUES (TIMESTAMP '2020-02-25 11:10:08.123');
ALTER SESSION SET TIME_ZONE='+08:00';
SELECT SESSIONTIMEZONE, ltzcol FROM LocalTZ;
```

返回结果：

```
+-----------------+-----------------------------+
| SESSIONTIMEZONE | ltzcol |
+-----------------+-----------------------------+
| +08:00 | 25-FEB-20 11:10:08.123000 AM |
+-----------------+-----------------------------+
```

可通过更改自定义参数 **TIME_ZONE** 的值而改变 **SESSIONTIMEZONE**，以下语句使用 TO_CHAR 函数指定输出格式：

```
ALTER SESSION SET TIME_ZONE='+00:00';
SELECT SESSIONTIMEZONE, TO_CHAR(ltzcol, 'YYYY-MM-DD HH:MI:SSXFF PM TZH:TZM') ltzcol FROM LocalTZ;
```

返回结果：

```
+-----------------+-------------------------------------+
| SESSIONTIMEZONE | ltzcol |
+-----------------+-------------------------------------+
| +00:00 | 2020-02-25 03:10:08.123000 AM +00:00 |
+-----------------+-------------------------------------+
```

4.1.15 INTERVAL YEAR TO MONTH 数据类型

与 DATE 和 TIMESTAMP 数据类型储存特定的时间点日期不同，INTERVAL YEAR TO MONTH 使用日期元素 YEAR 和 MONTH 来储存一段时间，此数据类型对于表示两个日期时间值之间的差异很有用。

语法

```
INTERVAL YEAR [(precision)] TO MONTH
```

参数

参数	值	说明
precision	0 ~ 9	代表了 YEAR 元素的精度，默认值为 2。也就是说在不指定该参数的情况下，最大可以存储 99 年 11 个月的间隔日期（最大不能超过 100 年）。如果存储的参数超过了默认精度 2 位，那么这个参数的值不能为空，必须被明确指定。

格式

插入 INTERVAL YEAR TO MONTH 数据类型的值时，有以下几种格式，更多关于间隔数据类型值的指定请参阅 间隔字面量 ：

语法	示例	说明
INTERVAL ‘year-month’ YEAR(percision) TO MONTH	INTERVAL ‘120-3’ YEAR(3) TO MONTH	间隔 120 年 3 个月。由于 YEAR 元素的值大于默认精度 2，所以必须指定 YEAR 元素的精度为 3.
INTERVAL ‘year’ YEAR(percision)	INTERVAL ‘50’ YEAR	间隔 50 年。
INTERVAL ‘month’ MONTH	INTERVAL ‘500’ MONTH	间隔 500 个月或者间隔 41 年 8 个月。

示例

如下代码所示，在表 **Interval_Sample** 中创建了数据类型为 INTERVAL YEAR TO MONTH 的三列 **interval1**、**interval2** 和 **interval3**，并向其中插入数值：

```
CREATE TABLE Interval_Sample (
interval1 INTERVAL YEAR TO MONTH,
interval2 INTERVAL YEAR(3) TO MONTH,
interval3 INTERVAL YEAR TO MONTH
);
INSERT INTO Interval_Sample (interval1, interval2, interval3)
VALUES (INTERVAL '12-3' YEAR TO MONTH, INTERVAL '120-3' YEAR(3) TO MONTH, INTERVAL '40' MONTH);
SELECT * FROM Interval_Sample;
```

返回结果：

```
+-----------+-----------+-----------+
| interval1 | interval2 | interval3 |
+-----------+-----------+-----------+
| +12-03 | +120-03 | +03-04 |
+-----------+-----------+-----------+
```

间隔与其他日期类型的计算

OceanBase 支持数据类型间的转换，所以间隔数据类型的值可以与其他日期值进行数学运算，但是数据库并不支持数据类型间任意的进行加、减、乘、除运算。请参阅文档 日期时间 DATE 和间隔 INTERVAL 的计算 中查看目前支持的日期类型计算矩阵图和参阅文档 数据类型转换 了解数更多数据类型转换的信息。

示例 1：间隔与间隔的计算，返回的值仍为间隔数据类型。

```
SELECT INTERVAL '2-2' YEAR TO MONTH -INTERVAL '1-1' YEAR TO MONTH calculate1, INTERVAL '2-2' YEAR TO MONTH + INTERVAL '1-1' YEAR TO MONTH calculate2 FROM DUAL;
```

返回结果：

```
+---------------+----------------+
| calculate1 | calculate2 |
+-------------------------------+
| +000000001-01 | +000000003-03 |
+---------------+----------------+
```

示例 2：间隔与日期时间的计算，返回的值为日期数据类型。

SYSDATE 返回当前的时间 **2020-02-27 16:13:50**，以下示例会返回从现在起两个月后的日期值。数据库只支持间隔+日期时间，而间隔-日期时间是无效运算。但是日期时间+间隔和日期时间-间隔都是有效运算。

```
SELECT TO_CHAR(INTERVAL '2' MONTH +SYSDATE,'YYYY-MM-DD HH24:MI:SS') calculate3 FROM DUAL;
```

返回结果：

```
+---------------------+
| calculate3 |
+---------------------+
| 2020-04-27 16:13:50 |
+---------------------+
```

示例 3：间隔和数字的运算，返回的值仍为间隔数据类型。间隔可以与数字进行乘除运算。以下示例展示了间隔 2 月乘以 2 的计算和间隔 2 天除以 3 的计算。

```
SELECT INTERVAL '2' MONTH*2 calculate4, INTERVAL '2' DAY/3 calculate5 FROM DUAL;
```

返回结果为间隔 4 月和间隔 16 小时：

```
+---------------+------------------------------+
| calculate4 | calculate5 |
+---------------+------------------------------+
| +000000000-04 | +000000000 16:00:00.000000000 |
+---------------+------------------------------+
```

更多信息

- 间隔字面量
- 数据类型转换
- 日期时间和间隔的计算

4.1.16 INTERVAL DAY TO SECOND 数据类型

INTERVAL DAY TO SECOND 可存储以天、小时、分钟和秒为单位的时间段。此数据类型对于表示两个日期时间值之间的精确差异很有用。

语法

```
INTERVAL DAY [(precision)] TO SECOND [(fractional_seconds_precision)]
```

参数

参数	值	说明
precision	0~9	代表了 DAY 元素的精度，默认值为 2。
fractional_seconds_precision	0~9	代表了 SECOND 元素小数部分的精度，默认值为 6。

示例

在插入 INTERVAL DAY TO SECOND 数据类型的值时，有以下几种格式，更多关于间隔数据类型值的指定请参阅间隔字面量 ：

语法	示例	说明
INTERVAL ‘dd hh:mm:ss’ DAY(precision) TO SECOND(fractional_seconds_precision)	INTERVAL ‘140 5:12:10.2222222’ DAY(3) TO SECOND(7)	间隔 140 天 5 小时 12 分钟 10.2222222 秒。
INTERVAL ‘dd hh’ DAY(precision) TO HOUR	INTERVAL ‘400 5’ DAY(3) TO HOUR	间隔 400 天 5 小时。
INTERVAL ‘dd hh:mm’ DAY(precision) TO MINUTE	INTERVAL ‘4 5:12’ DAY TO MINUTE	间隔 4 天 5 小时 12 分钟。
INTERVAL ‘hh:mm’ HOUR TO MINUTE	INTERVAL ‘11:20’ HOUR TO MINUTE	间隔 11 小时 20 分钟。
INTERVAL ‘hh:mm:ss’ HOUR TO SECOND(fractional_seconds_precision)	INTERVAL ‘11:12:10.2222222’ HOUR TO SECOND(7)	间隔 11 小时 12 分钟 10.2222222 秒。
INTERVAL ‘dd’ DAY(precision)	INTERVAL ‘14’ DAY	间隔 14 天。
INTERVAL ‘hh’ HOUR	INTERVAL ‘160’ HOUR	间隔 160 小时。
INTERVAL ‘mm’ MINUTE	INTERVAL ‘14’ MINUTE	间隔 14 分钟。
INTERVAL ‘ss’ SECOND(fractional_seconds_precision)	INTERVAL ‘14.666’ SECOND(2, 3)	间隔 14.666 秒。

如下代码所示，在表 **Interval_Sample** 中创建了数据类型为 INTERVAL DAY TO SECOND 的两列**interval1**、**interval2** 并向这两列中插入数值：

```
CREATE TABLE Interval_Sample (interval1 INTERVAL DAY TO SECOND, interval2 INTERVAL DAY(3) TO SECOND(3));
INSERT INTO Interval_Sample (interval1, interval2) VALUES ( INTERVAL '15 06:10:08' DAY TO SECOND, INTERVAL
'150 06:10:08' DAY(3) TO SECOND(3));
SELECT * FROM Interval_Sample;
```

返回结果：

```
+--------------------+------------------+
| interval1 | interval2 |
+--------------------+------------------+
| +15 06:10:08.000000 | +150 06:10:08.000 |
+--------------------+------------------+
```

间隔与其他日期类型的计算

OceanBase 支持数据类型间的转换，所以间隔数据类型的值可以与其他日期值进行数学运算。但是数据库并不支持数据类型间任意的进行加、减、乘、除运算。请参阅文档 日期时间 DATE 和间隔 INTERVAL 的计算 中查看目前支持的日期类型计算矩阵图和参阅文档 数据类型转换 了解数更多数据类型转换的信息。

关于间隔与其他数据类型计算的示例，请参阅文档 INTERVAL YEAR TO MON 数据类型 。

更多信息

- 间隔字面量
- 数据类型转换
- 日期时间和间隔的计算

4.1.17 日期时间 DATE 和间隔 INTERVAL 的计算

您可以在日期（DATE）、时间戳（TIMESTAMP、TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE）和间隔（INTERVAL YEAR TO MONTH 和 INTERVAL DAY TO SECOND）上执行许多算术运算。

OceanBase 根据以下规则计算结果：

- 您可以在日期和时间戳值（而非间隔值）的算术运算中使用 NUMBER 常数。OceanBase 在内部将时间戳值转换为日期值，并将算术日期时间和间隔表达式中的 NUMBER 常量解释为天数。例如，SYSDATE +1 是明天。SYSDATE-7 是一周前。SYSDATE +（10/1440）是十分钟之后。
  注意： 您不能乘以或除以日期或时间戳值。
- OceanBase 将 BINARY_FLOAT 和 BINARY_DOUBLE 操作数转换为 NUMBER。
- 每个 DATE 值都包含一个时间成分，许多日期运算的结果都包含一个分数。此分数表示一天的一部分。例如，1.5 天是 36 个小时。OceanBase 内置函数还返回这些分数，以对 DATE 数据执行常见操作。例如，MONTHS_BETWEEN 函数返回两个日期之间的月数。结果的小数部分代表一个 31 天月份的那部分。
- 如果一个操作数是 DATE 值或数字值，但都不包含时区或小数秒部分，则：
  - OceanBase 将其他操作数转换为 DATE 数据。数值乘以间隔例外，这将返回一个间隔。
    - 如果另一个操作数具有时区值，则 OceanBase 在返回的值中使用当前会话时区。
    - 如果另一个操作数具有小数秒数值，则小数秒数值将丢失。
- 当您将时间戳，时间间隔或数字值传递给仅限于 DATE 数据类型使用的内置函数时，OceanBase 将把非 DATE 值转换为 DATE 值。
- 当间隔计算返回日期时间值时，结果必须是实际的日期时间值，否则数据库将返回错误。例如，以下

两个语句返回错误：

```
SELECT TO_DATE("31-AUG-2004','DD-MON-YYYY") + TO_YMINTERVAL("0-1") FROM DUAL;
SELECT TO_DATE("29-FEB-2004','DD-MON-YYYY") + TO_YMINTERVAL("1-0") FROM DUAL;
```

第一条语句失败，因为在一个有31天的月份上增加一个月，导致计算结果为 9 月 31 日，这不是有效的日期。 第二条语句失败，因为在仅每四年存在的日期上增加一年是无效的。但是，在 2 月 29 日之前加上四年是有效的，计算结果为2008年2月29日：

```
SELECT TO_DATE("29-FEB-2004","DD-MON-YYYY") + TO_YMINTERVAL("4-0") FROM DUAL;
```

- OceanBase 以 GMT 时间执行所有时间戳算法。对于 TIMESTAMP WITH LOCAL TIME ZONE ，OceanBase 在执行算术运算时将日期时间值从数据库时区转换为 GMT，完成计算后再转换回数据库时区。对于 TIMESTAMP WITH TIME ZONE，日期时间值始终为 GMT，因此无需进行转换。
- 下表是日期时间算术运算的矩阵。 “—”表示不支持的操作。

数据类型	运算符	日期（DATE）	时间戳（TIMESTAMP）	间隔（INTERVAL）	数字（NUMERIC）
日期（DATE）	+	—	—	日期	日期
	-	数字	间隔	日期	日期
	*	—	—	—	—
	/	—	—	—	—
时间戳（TIMESTAMP）	+	—	—	时间戳	日期
	-	间隔	间隔	时间戳	日期
	*	—	—	—	—
	/	—	—	—	—
间隔（INTERVAL）	+	日期	时间戳	间隔	—
	-	—	—	间隔	—
	*	—	—	—	间隔
	/	—	—	—	间隔
数字（NUMERIC）	+	日期	日期	—	NA
	-	—	—	—	NA
	*	—	—	间隔	NA
	/	—	—	—	NA

4.1.18 RAW 数据类型

RAW 是一种可变长度的数据类型，在不同平台上传输时，传送的都是二进制信息，即使字符集不同也不需要转换。OceanBase 中用于保存二进制数据或字节字符串。

语法

```
RAW(length)
```

参数

参数	说明
length	表示长度，以字节为单位，作为数据库列最大存储 2000 字节的数据，作为变量最大存储 2000 字节的数据。

示例

示例 1：在 **test_raw** 表中声明 RAW 型数据，并向表中插入一条数据。

```
CREATE TABLE test_raw (c1 RAW(10));
INSERT INTO test_raw VALUES (utl_raw.cast_to_raw('1234567890'));
```

执行以下语句：

```
SELECT utl_raw.cast_to_varchar2(c1) FROM test_raw;
```

查询结果如下：

```
+------------------------------+
| UTL_RAW.CAST_TO_RAW(C1) |
+------------------------------+
| 1234567890 |
+------------------------------+
```

示例 2：向 **raw_test** 表中插入 2 条数据。

```
CREATE TABLE raw_test (id number, raw_date raw(10));
INSERT INTO raw_test VALUES (1, hextoraw('ff'));
INSERT INTO raw_test VALUES (2, utl_raw.cast_to_raw('051'));
```

执行以下语句：

```
SELECT * FROM raw_test;
```

查询结果如下：

```
+-----------+----------+
| ID | RAW_DATE |
+-----------+----------+
| 1 | ff |
+-----------+----------+
| 2 | 303531 |
+-----------+----------+
```

SQL 函数 HEXTORAW() 会把字符串中数据转换为 16 进制数。SQL 函数 UTL_RAW.CAST_TO_RAW([VARCHAR2]) 会把字符串中每个字符的 ASCII 码存放到 RAW 类型的字段中。例如，**051** 转换为 **303531**。

示例 3：OceanBase 将 RAW 数据转换为字符数据，每个字符代表 RAW 数据的四个连续位的十六进制数（0~9 和 A~F 或 a~f）。

如：二进制位数 11001011，其 RAW 数据转换后为字符 **CB**。

OceanBase 将字符数据转换为 RAW，它是把每个连续的输入字符解释为二进制数据的四个连续位，通过把这些位进行级联，来构建结果 RAW 值。

注意：如果任一输入字符不是十六进制数（0~9 和 A~F 或 a~f），则将报告错误。如果字符数为奇数，则结果

不确定。

更多信息

字符转换

在数据库之间传输数据，或者在数据之间传输数据库字符集和客户端字符集时，OceanBase 会自动在不同数据库字符集之间转换 CHAR 和 VARCHAR2 数据。而传输 RAW 数据时 OceanBase 不执行字符转换。

RAW 的函数

函数	说明
HEXTORAW()	当使用 hextoraw 时，会把字符串中数据转换成 16 进制串，字符串中的每两个字符表示了结果 RAW 中的一个字节。
RAWTOHEX(rawvalue)	将 RAW 类数值 rawvalue 转换为一个相应的十六进制表示的字符串。rawvalue 中的每个字节都被转换为一个双字节的字符串。
UTL_RAW_CAST_TO_RAW([VARCHAR2])	保持数据的存储内容不变，仅改变数据类型，将 VARCHAR2 转换为 RAW 类型。
UTL_RAW_CAST_TO_VARCHAR2([RAW])	保持数据的存储内容不变，仅改变数据类型，将 RAW 类型转换为 VARCHAR2 类型。
UTL_RAW.BIT_OR()、UTL_RAW.BIT_AND()、UTL_RAW.BIT_XOR()	位操作。

4.1.19 BLOB 数据类型

BLOB 全称为二进制大型对象（Binary Large Object）。它用于存储数据库中的大型二进制对象，可以将 BLOB 对象视为没有字符集语义的位流。BLOB 存储的二进制数据，其字节的长度上限为 48M，字符集是 BINARY。

BLOB 对象具有完整的事务支持。通过 SQL、DBMS_ LOB 软件包进行的更改将完全参与事务。可以提交和回滚 BLOB 值操作。但是，您不能在一个事务中将 BLOB 定位器保存在 PL/SQL 中，然后在另一事务或会话中使用它。

注意：存储的二进制文件过大，会使数据库的性能下降。

在数据库中，通常像图片、文件、音乐等大文件信息就用 BLOB 字段来存储，它先将大文件转为二进制再存储进去。

如下，创建表 **blob_table**，并设置 **blob_cl** 列为 BLOB 数据类型。

```
CREATE TABLE blob_table (blob_cl BLOB);
```

4.1.20 CLOB 数据类型

CLOB 全称为字符大型对象（Character Large Object）。它用于存储单字节和多字节字符数据。支持固定宽度和可变宽度字符集，且都使用数据库字符集。CLOB 不支持宽度不等的字符集。可存储字节的长度上限（字符）是 48 M，字符集是 UTF8MB4。

CLOB 对象具有完整的事务支持。通过 SQL、DBMS_ LOB 软件包进行的更改将完全参与事务。可以提交和回滚 CLOB 值操作。但是，您不能在一个事务中将 CLOB 定位器保存在 PL/SQL，然后在另一事务或会话中使用它。

由于 VARCHAR2 类型字段长度最大 32767，若需要保存的字段长度大于 32767，可以使用 CLOB 类型。另外，可以使用 CLOB 来保存 CHAR 数据，如 XML 文档就是用 CLOB 数据保存内容

例如，创建表 **temp**，设置 **temp_clob** 列为 CLOB 数据类型。

```
CREATE TABLE temp (temp_clob CLOB);
```

。

4.2 数据类型比较规则

4.2.1 数据类型比较规则概述

数据类型比较规则规定了 OceanBase 数据库如何比较每种数据类型的值。

OceanBase 数据类型比较规则支持以下数据值：

- 数值
- 日期值
- 字符值

更多信息

- 数据类型优先级
- 数据类型转换
- 数据转换的安全注意事项

4.2.2 数值

数值数据有定点数、浮点数和零。

数值数据比较规则如下：

- 较大的值大于较小的值，如：5.5 > 2.1。
- 负数均小于零，如 -3 < 0, -200 < -1。
- 正数均大于零，如 20 > 0, 100 > 1。

4.2.3 日期值

DATE 数据类型存储日期和时间信息。每个 DATE 值，OceanBase 存储以下信息：年、月、日、小时、分钟和秒，但是并不包含时区信息。

日期数据比较规则如下：

- 现在的时间大于过去的时间，如 2018 年 5 月 1 日的日期值大于 2012 年 5 月 1 日的日期值。
- 下午的时间大于早上的时间，如 2019 年 2 月 2 日下午 3:30 的日期时间值大于 2019 年 2 月 2 日

上午 10:30 的日期时间值。

- 早上的时间大于昨天的时间，如 2019 年 3 月 5 日上午 2:30 的日期时间值大于 2019 年 3 月 4 日下午 23:30 的日期时间值。

4.2.4 字符值

字符数据是根据字符值大小比较的，而字符值则根据以下两种度量进行比较：

- 二进制和语言比较
- 空白填充或非填充比较语义

二进制和语言比较

二进制比较

在默认的二进制比较中，OceanBase 根据数据库字符集中字符的数字代码的级联值比较字符串。如果一个字符在字符集中的数值大于另一个，则该字符更大。OceanBase 不支持 ASCII 字符集和 EBCDIC 字符集。在 OceanBase 的字符集中，规定空格小于任何字符的字符集：

- UTF-8
- UTF-16
- GBK
- GB18030

语言比较

在语言排序中，SQL 排序和比较都按照 NLS_SORT 指定的语言规则。若数字代码的二进制序列与要比较字符的语言序列不匹配，则使用语言比较。若 NLS_SORT 参数的设置不是 BINARY，且 NLS_COMP 参数设置为 LINGUISTIC，则使用语言比较。

空白填充和非填充比较语义

空白填充比较语义

使用空白填充语义，若两个值的长度不同，则 OceanBase 首先将空格添加到较短的空格的末尾，以便它们的长度相等。然后，OceanBase 逐个字符地比较值，直到第一个不同的字符为止。在第一个不同位置具有较大字符的值被认为较大。如果两个值没有不同的字符，则认为它们相等。此规则意味着两个值仅在尾随空白数上不同时相等。

注意：只有当比较中的两个值都是数据类型 CHAR、NCHAR、文本文字或 USER 函数返回的值时，OceanBase 就会使用空白填充的比较语义。

非填充比较语义

使用非填充语义，OceanBase 逐个字符地比较两个值，直到第一个不同的字符为止。该位置上具有较大字符的值被认为较大。如果两个不同长度的值在较短的值之前一直相同，则较长的值被认为较大。如果两个长度相等的值没有不同的字符，则认为这些值相等。

注意：只要比较中的一个或两个值的数据类型为 VARCHAR2 或 NVARCHAR2，OceanBase 就会使用非填充比较语义。

示例

使用不同的比较语义比较两个字符值的结果不同。本示例显示了使用空白填充语义和非填充语义比较。

空白填充	非填充
'ac' > 'ab'	'ac' > 'ab'
'ab' > 'a '	'ab' > 'a '
'ab' > 'a'	'ab' > 'a'
'ab' = 'ab'	'ab' = 'ab'
'ac' > 'ab'	'ac' > 'ab'
'a ' = 'a'	'a ' > 'a'

通常，空白填充和非填充比较的结果是相同的。而最后一行的比较示例说明了空白填充和非填充比较语义之间的区别。

4.2.5 数据类型优先级

OceanBase 使用数据类型优先级来确定隐式数据类型转换顺序。

OceanBase 数据类型的转换优先级如下（由高到低）：

1. 日期时间和间隔数据类型
2. BINARY_DOUBLE 数据类型
3. BINARY_FLOAT 数据类型
4. NUMBER 数据类型
5. 字符数据类型
6. 所有其他内置数据类型

4.2.6 数据类型转换

通常，表达式不能包含不同数据类型的值。但是为了使表达式能够进行计算，OceanBase 支持从一个数据类型到另一个数据类型的值的 隐式转换 和 显式转换 。

隐式数据类型转换

当转换有意义时，OceanBase 数据库会自动将一个值从一种数据类型转换为另一种数据类型。隐式数据类型的转换规则：

- INSERT 和 UPDATE 操作时，OceanBase 把变量值转换成列类型。
- SELECT FROM 操作时，OceanBase 把列数据类型转换成目标变量类型。
- 字符值和数字值比较时，OceanBase 把字符值转换成数字值。

- 在处理数值时，OceanBase 会调整精度和小数位数。由此产生的数字数据类型与基础表中找到的数字数据类型不同。
- 字符值或数值和浮点数值之间的转换可以是不精确的，因为字符类型和数量使用十进制精度来表示数值，浮点数使用二进制精度。
- 当一个 CLOB 值转换为一个字符类型如 VARCHAR2，或 BLOB 转换为 RAW 时。如果要转换的数据大于目标数据类型，那么数据库会返回一个错误。
- 在从时间戳值转换为 DATE 值的过程中，时间戳值的小数秒部分被截断，且时间戳值的小数秒部分进行四舍五入。
- 从 BINARY_FLOAT 转换为 BINARY_DOUBLE 是准确的。
- 如果 BINARY_DOUBLE 的精度位数超出了 BINARY_FLOAT 支持的位数，BINARY_DOUBLE 转换为 BINARY_FLOAT 是不精确的。
- 当将字符值与 DATE 值进行比较时，OceanBase 将字符数据转换为 DATE。
- 赋值操作时，OceanBase 把等号右边的值转换成左边赋值目标数据类型。
- 连接操作时，OceanBase 把非字符类型转换成字符类型或国家字符类型。

隐式数据类型转换矩阵

下表显示了所有的隐式数据类型转换，您不需要考虑转换的方向或转换的上下文。 “-”表示不支持转换。

数据类型	CHAR	VARCHAR2	NCHAR	NVARCHAR2	DATE	DATETIME / INTERVAL	NUMBER	BINARY_FLOAT	BINARY_DOUBLE	RAW	CLOB	BLOB
CHAR	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
VARCHAR2	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
NCHAR	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
NVARCHAR2	Yes	Yes	Yes	-	Yes	Yes	Yes	Yes	Yes	Yes	Yes	-
DATE	Yes	Yes	Yes	Yes	-	-	-	-	-	-	-	-
DATETIME / INTERVAL	Yes	Yes	Yes	Yes	-	-	-	-	-	-	-	-
NUMBER	Yes	Yes	Yes	Yes	-	-	-	Yes	Yes	-	-	-
BINARY_FLOAT	Ye	Yes	Yes	Yes	-	-	Yes	-	Yes	-	-	-

	s											
BINARY_DOUBLE	Yes	Yes	Yes	Yes	-	-	Yes	Yes	-	-	-	-
RAW	Yes	Yes	Yes	Yes	-	Yes 1	-	-	-	-	Yes	-
CLOB	Yes	Yes	Yes	Yes	-	-	-	-	-	-	-	Yes
BLOB	-	-	-	-	-	-	-	-	-	Yes	-	-

1 您不能直接将 RAW 转换为 INTERVAL，但是可以使用 UTL_RAW.CAST_TO_VARCHAR2([RAW]) 将RAW 转换为 VARCHAR2，然后将所得的 VARCHAR2 值转换为 INTERVAL。

不同字符类型之间隐式转换的方向

数据类型	TO_CHAR	TO_VARCHAR2	TO_NCHAR	TO_NVARCHAR2
from CHAR	-	VARCHAR2	NCHAR	NVARCHAR2
from VARCHAR2	VARCHAR2	-	NVARCHAR2	NVARCHAR2
from NCHAR	NCHAR	NCHAR	-	NCHAR2
from NVARCHAR2	NVARCHAR2	NVARCHAR2	NVARCHAR2	-

隐式数据类型转换示例

执行以下语句：

```
SELECT 5 * 10 + 'james' FROM DUAL;
```

语句执行失败，且您收到以下报错：

```
invalid number
```

这是由于 OceanBase 使用了隐式数据类型转换，将 'james' 转换为数字类型，但是转换失败。

本示例将字符串 **‘2’** 从 CHAR 数据类型隐式转换为了数字数据类型 **2**，计算结果为 **52**。

执行以下语句：

```
SELECT 5 * 10 + '2' FROM DUAL;
```

查询结果如下：

```
+--------------------+
| 5 * 10 + '2' |
+--------------------+
| 52 |
+--------------------+
```

显式数据类型转换

您可以使用 SQL 转换函数转换数据类型，SQL 函数显式转换一个数据类型为另一个数据类型。

显示类型转换矩阵

数据类型	To CHAR、VARCHAR2、NCHAR、NVARCHAR2	TO_NUMBER	To Datetime/Interval	TO_RAW	TO_CLOB、BLOB	To_BINARY_FLOAT	To_BINARY_DOUBLE
From CHAR、VARCHAR2、NCHAR、NVARCHAR2	TO_CHAR(char .)、TO_NCHAR(char .)	TO_NUMBER	TO_DATE、TO_TIMESTAMP、TO_TIMESTAMP_TZ、TO_YMINTERVAL、TO_DSINTERVAL	HEXTORAW	TO_CLOB	TO_BINARY_FLOAT	TO_BINARY_DOUBLE
From NUMBER	TO_CHAR(number)、TO_NCHAR(number)	—	TO_DATE、NUMTOYM_INTERVAL、NUMTOOLS_INTERVAL	—	—	TO_BINARY_FLOAT	TO_BINARY_DOUBLE
From Datetime/Interval	TO_CHAR(date)、TO_NCHAR(date)	—	—	—	—	—	—
From RAW	RAWTOHEX、RAWTONHEX	—	—	—	TO_BLOB	—	—
From CLOB、BLOB	TO_CHAR、TO_NCHAR	—	—	—	TO_CLOB	—	—
From BINARY_FLOAT	TO_CHAR(char .)、TO_NCHAR(char .)	TO_N	—	—	—	TO_BINARY_FLO	TO_BINARY_DOUBLE

		UMBER				AT	
From BINARY_DOUBLE	TO_CHAR(char.)、TO_NCHAR(char.)	TO_NUMBER	—	—	—	TO_BINARY_FLOAT	TO_BINARY_DOUBLE

显式数据类型转换示例

当前的时间通过 TO_CHAR 函数显式转换为想要的格式输出。执行以下语句：

```
SELECT TO_CHAR(SYSDATE, 'YYYY_MM_DD') FROM DUAL;
```

查询结果如下：

```
+-------------------------------+
| TO_CHAR(SYSDATE,'YYYY_MM_DD') |
+-------------------------------+
| 2020_02_27 |
+-------------------------------+
```

4.2.7 数据转换的安全注意事项

通过隐式转换或不指定格式模型的显式转换将日期时间值转换为文本时，格式模型由一个全局会话参数定义。根据源数据类型，这些参数名称为 NLS_DATE_FORMAT、NLS_TIMESTAMP_FORMAT 或 NLS_TIMESTAMP_TZ_FORMAT。这些参数的值可以在客户端环境或 ALTER SESSION 语句中指定。

当不指定格式模型的显式转换对动态构造的 SQL 语句中的日期时间值进行显式转换时，格式模型对会话参数的选择过程会对数据库安全性产生负面影响。

动态构造的 SQL 语句是指由程序或者存储过程生成的 SQL 语句。执行动态构造的 SQL 语句，需要 OceanBase 内置的 PL/SQL 包 DBMS_SQL 或与 PL/SQL 语句 EXECUTE IMMEDIATE 相关，但这些并不是唯一执行动态构造的 SQL 文本的方式。

如下所示，start_date 的数据类型为 DATE，使用会话参数 NLS_DATE_FORMAT 中指定的格式模型将 start_date 的值转换为文本，再将结果传递到 SQL 文本中。日期时间格式模型可以简单地由双引号所包含的文本组成。

```
SELECT last_name FROM employees WHERE hire_date > '' || start_date || '';
```

说明：为显式转换的格式模型设置全球化参数的用户可以决定上述转换产生了什么文本。若 SQL 语句由过程执行，则该过程通过会话参数易受到对数据库进行增删改查的影响。存储过程分为 DR (Definer’ s Rights) Procedure 和 IR (Invoker’ s Rights) Procedure 两种。若该过程使用 Definer 的权限运行，且具有比会话本身更高的特权，则用户可以获得对敏感数据，未授权的访问。

4.3 字面量

4.3.1 字面量概述

字面量（Literals）是用于表达一个固定值的表示法。许多函数和 SQL 语句都需要指定字面量，字面量也可以作为表达式和条件的一部分。OceanBase 支持以下字面量：

- 文本字面量
- 数值字面量
- 日期字面量
- 间隔字面量

文本字面量

文本字面量是指使用单引号 ' ' 引起来的字符串。

数值字面量

数值字面量是用来指定固定数和浮点数的值。此类别中有以下两种字面量：

字面量	说明
整数 INTEGER 字面量	当表达式、条件、SQL 函数和 SQL 语句中出现整数时，需要用整数字面量来指定值。
数字 NUMBER 和浮点数 Floating-Point 字面量	当表达式、条件、SQL 函数和 SQL 语句中出现数字时，需要用数字或浮点字面量来指定值。

日期时间字面量

用来指定代表日期和时间的值。此类别中有以下两种字面量：

字面量	说明
日期字面量	日期字面量可以通过字符串指定，或者可以使用 TO_DATE 函数将字符或数字值转换为日期值。
时间戳字面量	时间戳字面量里包含 TIMESTAMP 字面量、 TIMESTAMP WITH TIME ZONE 字面量和 TIMESTAMP WITH LOCAL TIME ZONE 字面量三种。可以指定包含年、月、日、时、分、秒和时区的时间戳值。

间隔字面量

间隔字面量用来指定一段时间的值。此类别中有以下两种字面量：

字面量	说明
INTERVAL YEAR TO MONTH 字面量	用来指定以年和月为单位的一段时间。
INTERVAL DAY TO SECOND 字面量	用来指定以天和具体时间为单位的一段时间。

更多信息

- 前导字段和尾随字段

4.3.2 文本字面量

文本字面量（Text Literals）是使用单引号 ' ' 引起来的字符串，用来在表达式、条件、SQL 函数、SQL 语句中指定字符串的值。

文本字面量本身具有 CHAR 和 VARCHAR2 数据类型的属性：

- 在表达式和条件中，OceanBase 通过使用空白填充的比较语义进行比较，将文本字面量视为数据类型为 CHAR。
- 指定文本字面量时，CHAR 数据类型的值的长度最大是 2000，VARCHAR2 数据类型的值的长度最大是 32767。

以下是一些有效的文本字面量，若要在字符串中表现一个单引号，需要在字符串中的单引号前再插入一个单引号：

```
'Jackie''s raincoat'<br>
'Hello'<br>
'09-MAR-98'<br>
'今天天气很好'
```

4.3.3 数值字面量

数值字面量（Numberic Literals）使用数值字面量指定固定数和浮点数的值。

整数字面量

当表达式、条件、SQL 函数和 SQL 语句中出现整数时，需要用整数（Integer）字面量来指定值。

下面是一些有效的整数字面量：

```
8
+186
-15
```

数字和浮点数字面量

当表达式、条件、SQL 函数和 SQL 语句中出现数字时，需要用数字（Number）或浮点数（Floating-Point）字面量来指定值。

以下是一些有效的数字字面量：

```
12
+6.87
0.5
25e-03
-9
```

以下是一些有效的浮点数字面量：

```
25f
+6.34F
0.5d
-1D
```

数字字面量最大可以储存精度为 38 位的数字。如果字面量要求的精度比 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 所提供的精度更高，则 OceanBase 将截断该值。如果字面量的范围超出 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 支持的范围，则 OceanBase 会抛出错误。

示例

数值字面量中的小数点分隔符始终是点（.）。在期望数值字面量的地方指定了文本字面量，则该文本字面量被转换为数值字面量。

在下面的示例中，计算了 **2** 乘以数值字面量 **2.2** 和 **2** 乘以文本字面量 **‘3.3’** ：

```
SELECT 2*2.2, 2*'3.3' FROM DUAL;
```

返回结果：

```
+-------+---------+
| 2*2.2 | 2*'3.3' |
+-------+---------+
| 4.4 | 6.6 |
+-------+---------+
```

4.3.4 日期字面量

日期字面量（Date Literals）可以通过字符串指定，或者可以使用 TO_DATE 函数将字符或数字值转换为日期值。日期字面量是唯一接受用 TO_DATE 表达式代替字符串指定值的：

```
TO_DATE('2020-03-25 11:05:00', 'YYYY-MM-DD HH24:MI:SS')
```

使用日期值指定日期字面量时，必须使用公历的日期值。同时也可以如下所示，使用 ANSI 来指定日期字面量，ANSI 日期字面量不包含时间信息，而且必须使用 **YYYY-MM-DD** 的格式：

```
DATE '2020-03-25'
```

此外，还可以使用数据库默认日期值来指定日期字面量，当在日期表达式中使用默认值时，OceanBase 会自动将默认日期格式的字符值转换为日期值。数据库的默认日期值由初始化参数 NLS_DATE_FORMAT 指定，此示例中默认格式为 **DD-MON-RR** ：

```
TO_DATE('25-FEB-20', 'DD-MON-RR')
```

如果您指定不带时间成分的日期值，则默认时间为午夜（24 小时制 00:00:00 和 12 小时制 12:00:00）。如果指定的日期值不带日期成分，则默认日期为当前月份的第一天。

OceanBase 表中 DATE 列始终同时包含日期和时间字段。因此，如果查询 DATE 列，则必须在查询中指定时间字段，或确保 DATE 列中的时间字段设置为午夜。否则，数据库可能不会返回您期望的查询结果。比如创建一张具有 **id** 列和 **datecol** 日期列的表 **Date_Literals**：

```
CREATE TABLE Date_Literals (id NUMBER, datecol DATE);
```

在表中插入当前会话的系统日期时间 **SYSDATE**，此示例使用了 TRUNC 函数将时间字段设置为午夜， TRUNC 函数会截取 **SYSDATE** 的日期部分，这样 **datecol** 列中的时间会自动填充默认的午夜时间：

```
INSERT INTO Date_Literals VALUES (1,SYSDATE);
INSERT INTO Date_Literals VALUES (2,TRUNC(SYSDATE));
```

此时表中数据为：

```
+------+--------------------+
| id | datecol |
+------+--------------------+
| 1 | 25-FEB-20 11:28:16 |
| 2 | 25-FEB-20 00:00:00 |
+------+--------------------+
```

当查询中不包含时间信息时，可以在查询中使用大于或小于条件，而不是等于或不等于条件：

```
SELECT * FROM Date_Literals WHERE datecol > TO_DATE('2020-02-24', 'YYYY-MM-DD');
```

返回结果：

```
+------+--------------------+
| id | datecol |
+------+--------------------+
| 1 | 25-FEB-20 11:28:16 |
| 2 | 25-FEB-20 00:00:00 |
+------+--------------------+
```

当使用等于条件时，由于查询中不包含时间信息，所以结果只返回了时间信息为午夜值的日期：

```
SELECT * FROM Date_Literals WHERE datecol = TO_DATE('2020-02-25', 'YYYY-MM-DD');
```

返回结果：

```
+------+--------------------+
| id | datecol |
+------+--------------------+
```

```
| 2 | 25-FEB-20 00:00:00 |
+------+--------------------+
```

反过来，可以过滤掉 **datecol** 列中的时间字段，只查询日期字段：

```
SELECT * FROM Date_Literals WHERE TRUNC(datecol) = DATE '2020-02-25';
```

返回结果：

```
+------+--------------------+
| id | detacol |
+------+--------------------+
| 1 | 25-FEB-20 11:28:16 |
| 2 | 25-FEB-20 00:00:00 |
+------+--------------------+
```

4.3.5 时间戳字面量

OceanBase 支持以下三种时间戳字面量（Timestamp Literals）：

- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE

TIMESTAMP 字面量

TIMESTAMP[(scale)] 数据类型存储了年、月、日、时、分、秒和小数秒值的值。当指定 TIMESTAMP 字面量时，秒字段最大可以指定精度到第9位的纳秒：

```
TIMESTAMP '2020-02-25 11:26:18.316'
```

TIMESTAMP WITH TIME ZONE 字面量

TIMESTAMP WITH TIME ZONE 字面量是包含时区信息的时间戳字面量。

TIMESTAMP [(scale)] WITH TIME ZONE 数据类型是 TIMESTAMP[(scale)] 数据类型的一种变体，它在 TIMESTAMP[(scale)] 数据类型的基础上还存储 时区偏移量或者时区区域名称 等信息。在指定 TIMESTAMP WITH TIME ZONE 字面量时，需要指定时区信息并且秒字段最大可以指定精度到第9位的纳秒，以下示例使用时区偏移量指定了时区字段的值：

```
TIMESTAMP '2020-02-25 11:26:18.316 +08:00'
```

当两个 TIMESTAMP WITH TIME ZONE 字面量中的值表示 GMT 时区的同一时刻，那么尽管它们的时区字段的值不同，它们也被视为相同的字面量。如以下示例所示，GMT-8 时区的早上 8 点和 GMT-5 时区的早上 11 点其实是同一时刻：

```
TIMESTAMP '2020-04-25 08:26:18.316 -08:00'
TIMESTAMP '2020-04-25 11:26:18.316 -05:00'
```

同样的，在字面量中我们可以使用时区区域名称替换时区偏移量，如下所示的示例中将 **-08:00** 替换为了 **America/Los_Angeles**：

```
TIMESTAMP '2020-02-01 11:00:00 America/Los_Angeles'
```

由于一些地区有 夏令时 的转换，为了消除夏令时转换时时间的歧义，在指点字面量的值时可以同时使用 时区区域名称（TZR）和相应的缩写 （TZD）以确保字面量的值为夏令时：

```
TIMESTAMP '2020-06-01 11:00:00 America/Los_Angeles PDT'
```

TIMESTAMP WITH LOCAL TIME ZONE 字面量

TIMESTAMP [(scale)] WITH LOCAL TIME ZONE 数据类型是包含本地时区信息的数据类型。OceanBase 中并没有专门的 TIMESTAMP WITH LOCAL TIME ZONE 字面量，是通过其他有效的日期时间字面量来为TIMESTAMP [(scale)] WITH LOCAL TIME ZONE 数据类型赋值的。下表显示了一些可用于将值插入 TIMESTAMP WITH LOCAL TIME ZONE 列的格式，以及查询返回的相应值：

INSERT 语句中指定的值	查询返回的值
‘25-FEB-20’	25-FEB-20 00.00.000000
SYSTIMESTAMP	25-FEB-20 14:28:41.264258
TO_TIMESTAMP(‘25-FEB-2020’, ‘DD-MON-YYYY’)	25-FEB-20 00.00.000000
SYSDATE	25-FEB-20 02.55.29.000000 PM
TO_DATE(‘25-FEB-20’, ‘DD-MON-YYYY’)	25-FEB-20 12.00.00.000000 AM
TIMESTAMP’ 2020-02-25 8:00:00 America/Los_Angeles’	25-FEB-20 08.00.00.000000 AM

4.3.6 间隔字面量

间隔字面量（Interval Literals）用来指定一段时间的值。OceanBase 支持两种类型的间隔字面量：

- INTERVAL YEAR TO MONTH
- INTERVA DAY TO SECOND

前导字段和尾随字段

每种间隔字面量都包含一个前导字段和一个可选的尾随字段。前导字段定义了要测量的日期或时间的基本单位，尾随字段定义了所考虑的基本单位的最小增量。例如，DAY TO MINUTE 用来指定最小单位到月份的间隔字面量，其中前导字段是 YEAR，尾随字段是 MINUTE。尾随字段是可选的，在指定间隔字面量时可以省去。

在间隔字面量中有以下字段：YEAR、MONTH、DAY、HOUR、MINUTE 和 SECOND。它们的权重从 YEAR 开始按顺序递减。当需要指定尾随字段时，字面量中尾随字段的权重一定要低于前导字段，否则是无效的指定。例如，INTERVAL '1-2' DAY TO YEAR 是个无效的字面量。

前导字段值的位数范围为 0~9，默认值为 2。SECOND 字段指定了秒数，该字段最大可以精确到小数点后 9 位，最小是小数点 0 位，默认值精度是小数点 6 位。字段的值超出指定的范围后，数据库会返回错误。SECOND 字段的小数位数如果超出指定精度，会四舍五入到符合指定精度的值。

INTERVAL YEAR TO MONTH 字面量

INTERVAL YEAR TO MONTH 字面量用来指定以年和月为单位的一段时间。

以下是一些 INTERVAL YEAR TO MONTH 字面量的示例：

示例	说明
INTERVAL ‘265-2’ YEAR(3) TO MONTH	间隔 265 年 2 个月。前导字段 YEAR 的精度大于默认的 2 位，需要指定与值位数相符的精度值
INTERVAL ‘265’ YEAR(3)	表示间隔 265 年。
INTERVAL ‘500’ MONTH(3)	表示间隔 500 个月或 41 年 8 个月。
INTERVAL ‘10’ MONTH	表示隔 10 个月。
INTERVAL ‘123’ YEAR	返回错误，值 123 超出了默认精度 2 位。

可以在一个 INTERVAL YEAR TO MONTH 字面量之间添加或减去另一个 INTERVAL YEAR TO MONTH 字面量。例如：INTERVAL '6-2' YEAR TO MONTH + INTERVAL'21' MONTH = INTERVAL '7-11' YEAR TO MONTH。

INTERVAL DAY TO SECOND 字面量

INTERVAL DAY TO SECOND 字面量用来指定以天和具体时间为单位的一段时间。

以下是一些 INTERVAL DAY TO SECOND 字面量的示例：

示例	说明
INTERVAL ‘4 5:12:10.222’ DAY TO SECOND(3)	表示间隔 4 天 5 小时 12 分钟 10.222 秒。SECOND 字段小数点默认精度是 6，这里如果不手动指定精度为 3，返回的结果中会用 0 补齐位数。
INTERVAL ‘4 5:12’ DAY TO MINUTE	表示间隔 4 天 5 小时 12 分钟
INTERVAL ‘400 5’ DAY(3) TO HOUR	表示间隔 400 天 5 小时。前导字段 DAY 超出默认精度 2 位，这里手动指定精度为 3。
INTERVAL ‘400’ DAY(3)	表示间隔 400 天。
INTERVAL ‘11:12:10.2222222’ HOUR TO SECOND(7)	表示间隔 11 小时 12 分钟 10.2222222 秒。SECOND 字段的值超出默认精度 6 位，这里手动指定了和值相符的精度。
INTERVAL ‘11:20’ HOUR TO MINUTE	表示间隔 11 小时 20 分钟。
INTERVAL ‘10’ HOUR	表示间隔 10 小时
INTERVAL ‘10:22’ MINUTE TO SECOND	表示间隔 10 分钟 22 秒。
INTERVAL ‘10’ MINUTE	表示间隔 10 分钟。
INTERVAL ‘4’ DAY	表示间隔 4 天。
INTERVAL ‘25’ HOUR	表示间隔 25 小时。
INTERVAL ‘40’ MINUTE	表示间隔 40 分钟。

INTERVAL ‘120’ HOUR(3)	表示间隔 120 小时。
INTERVAL ‘30.12345’ SECOND(2,4)	表示间隔 30.1235 秒。秒的小数点位数超出指定精度，所以四舍五入到小数点第 4 位。

可以在一个 INTERVAL DAY TO SECOND 字面量之间添加或减去另一个 INTERVAL DAY TO SECOND 字面量。例如：INTERVAL'20' DAY - INTERVAL'239' HOUR = INTERVAL'10-1' DAY TO SECOND。

4.4 格式化

4.4.1 格式化概述

格式化指定了存储在数据库中的日期时间数据或数值数据的格式。当您将字符串转换为日期时间或数字时，格式化会告诉 OceanBase 数据库如何转换并存储该字符串。在 SQL 语句中，您可以通过 TO_CHAR 、TO_NUMBER 和 TO_DATE 等函数的参数来指定：

- OceanBase 数据库返回值的格式
- 存储在数据库中值的格式

OceanBase 支持以下几种数据格式化：

数值格式化

数值格式化指定了存储在数据库中的定点数和浮点数的格式。当您需要将 SQL 语句中出现 NUMBER，BINARY_FLOAT 或 BINARY_ DOUBLE 值转换为 VARCHAR2 数据类型时，您可以使用函数中的数值格式化。数值格式化是由一个或多个数值格式化元素组成，具体信息请查阅 数值格式化 。

日期时间格式化

日期时间格式化指定了存储在数据库中日期时间数据的格式。日期时间格式化的总长度不能超过 22 个字符。当您需要将非默认格式的字符值转换为日期时间格式的值时，您可以使用函数中的日期时间格式化。日期时间格式化是由一个或多个日期时间格式化元素组成，具体信息请查阅 日期时间格式化 。将字符串值转换为日期值，是有一些转换规则，关于规则请查阅 字符串到日期的转换规则 。

RR 日期时间格式化元素类似于 YY 日期时间格式化元素，但它为跨世纪日期值存储提供了额外的灵活性，关于 RR 日式格式化元素，请查阅 RR 日期时间格式化元素 。

格式化修饰符

OceanBase 数据库暂不支持格式化修饰符FX和FM。

4.4.2 数值格式化

数值格式化指定了存储在数据库中的定点数和浮点数的格式。

函数中的数值格式化

以下数值数据类型转换函数使用了数值格式化：

当表达式、条件、SQL 函数和 SQL 语句中出现 NUMBER，BINARY_FLOAT 或 BINARY_ DOUBLE 时，且

您需要它们的值转换为 VARCHAR2 数据类型时，需要用 TO_CHAR 函数的参数指定这些数值的格式。

当表达式、条件、SQL 函数和 SQL 语句中出现 CHAR 或 VARCHAR2 时，如果您需要将他们的值转换为 NUMBER 数据类型时，需要用 TO_NUMBER 函数的参数（暂不支持 NLS_NUMERIC_CHARACTERS）指定这些数值的格式。如果您需要将他们的值转换为 BINARY_FLOAT 或 BINARY_DOUBLE 时，需要用 TO_BINARY_FLOAT 和 TO_BINARY_DOUBLE 函数的参数指定数值格式。

数值格式化会将数值四舍五入，并取有效数字位数。如果某个值的小数位数，左边的有效位数比格式中指定的位数高，则用 # 代替该值。如果 NUMBER 的正值非常大且无法以指定的格式表示，则使用无穷大符号（∽）替换该值。如果负 NUMBER 值非常小且无法用指定的格式表示，则负无穷大符号将替换值（-∽）。

数值格式化的元素

与 Oracle 不同，OceanBase 数值格式化元素仅支持标准的数值格式。下表为 OceanBase 支持的数值格式化元素：

元素	示例	说明
.（小数点）	99.99	返回一个小数，且小数点在指定位置。 **限制条件**：在数字格式化中，您只能指定一个小数点。
0	0999 9990	0999 返回前导零。 9990 返回尾随零。
9	9999	返回具有指定位数的值。如果为正，则返回带有有前导空格的数；如果为负，则返回前导负数。前导零返回 0，除了零值，定点数的小数部分返回零。

示例

执行以下语句：

```
SELECT TO_CHAR(0, '99.99') FROM DUAL;
```

查询结果如下：

```
+--------------------+
| TO_CHAR(0,'99.99') |
+--------------------+
| .00 |
+--------------------+
```

下表显示了对不同数值的 **number** 按照格式化元素 ‘fmt’ 查询得到的结果。

```
SELECT TO_CHAR(number, 'fmt') FROM DUAL;
```

number	‘fmt’	Result
0	99.99	‘ .00’
+0.1	99.99	‘ .10’
-0.2	99.99	‘ -.20’

0	90.99	‘ 0.00’
+0.1	90.99	‘ 0.10’
-0.2	90.99	‘ -0.20’
0	9999	‘ 0’
1	9999	‘ 1’
+123.456	999.999	‘ 123.456’
-123.456	999.999	‘-123.456’

4.4.3 日期时间格式化

日期时间格式化指定了存储在数据库中日期时间数据的格式。日期时间格式化的总长度不能超过 22 个字符。

函数中的日期时间格式化

日期时间格式化出现在下面的数据类型转换中：

将非默认格式的字符值转换为日期时间值时，需要 TO_DATE、TO_TIMESTAMP 和 TO_TIMESTAMP_TZ 函数的参数指定日期时间的格式。

将日期时间值转换为非默认格式的字符值时，需要指定 TO_CHAR 函数的参数。

您可以通过以下方式指定日期时间格式。

- 通过会话参数 NLS_DATE_FORMAT、NLS_TIMESTAMP_FORMAT 或 NLS_TIMESTAMP_TZ_FORMAT 显式指定。
- 通过会话参数 NLS_TERRITORY 隐式指定。
- ALTER SESSION 语句更改会话的默认日期时间格式。

日期时间格式化

日期时间格式化由一个或多个日期时间格式化元素组成。OceanBase 支持的格式化元素请查阅 日期时间格式化的元素 。

- 在格式化字符串中，相同的格式化元素不能出现两次，表示类似信息的格式化元素不能组合。例如，您不能在一个格式化字符串中同时使用 SYYYY 和 BC 元素。
- 所有格式化元素都可以在TO_CHAR、TO_DATE、TO_TIMESTAMP和TO_TIMESTAMP_TZ函数中使用。
- 日期时间格式化元素FF、TZD、TZH、TZM 和 TZR可以出现在时间戳和间隔格式中，但不能出现在DATE格式中。
- 许多日期时间格式元素被空白填充或用零填充至指定的长度。

注意： OceanBase 建议您使用 4 位数的年份元素（YYYY），较短的年份元素会影响查询优化，因为年份只能在运行时确定。

日期时间格式化元素表

元素	日期时间函数是否支持？	说明
- / , . ; : “文字”	是	标点和引用的文本会在结果中复制。
AD A.D.	是	表示公元纪年法，带有或不带有点号。
AM A.M.	是	表示上午，带有或不带有点号。
BC B.C.	是	表示公元前的年份，带有或不带有点号。
D	是	星期几（1-7）。
DAY	是	一天的名称。
DD	是	每月的一天（1-31）。
DDD	是	一年中的某天（1-366）。
DL	是	只能打印类似 “Monday, January, 01, 1996” 的固定格式。
DS	是	只能打印类似 “10-10-1996” 的固定格式。
DY	是	日期的缩写，返回星期值。
FF [1..9]	是	小数秒。使用数字 1~9 来指定返回值的小数秒部分的位数。默认为日期时间数据类型指定的精度。在时间戳和间隔格式中有效，但在 DATE 格式中无效。
FX	是	需要字符数据和格式模型之间的精确匹配。
HH HH12	是	小时（1-12）。12 小时制
HH24	是	小时（0-23）。24 小时制
YYYY	是	包含4位数字的年份。
MI	是	分钟（0-59）。
MM	是	月（01-12；一月份表示为 01 ）。
MON	是	月份的缩写。
MONTH	是	月份名称。
PM P.M.	是	表示下午，带有或不带有点号。
Q	是	季度（1、2、3、4； 1月-3月是第 1 季度）。
RR	是	RR 匹配两位数的年份。
RRRR	是	年。接受4位或2位输入。
SS	是	秒（0-59）。
SSSSS	是	午夜后的秒（0-86400）。
TZD	是	夏令时信息。TZD 值是带有夏令时信息的缩写时区字符串。在时间戳和间隔格式中有效，但在 DATE 格式中无效。
TZH	是	时区小时。在时间戳记和间隔格式中有效，但在 DATE 格式中无效。
TZM	是	时区分钟。在时间戳记和间隔格式中有效，但在 DATE 格式中无效。

TZR	是	时区区域信息。在时间戳和间隔格式中有效，但在 DATE 格式中无效。
X	是	小数点，永远是 ‘.’ 。
Y,YYY	是	带逗号的年。
YYYY SYYYY	是	4位数字的年份。 S 代表用一个负号表示公元前的日期。
YYY YY Y	是	年份的后 3、2 或 1 位数字。

说明：日期时间函数指的是 TO_CHAR、TO_DATE、TO_TIMESTAMP 和 TO_TIMESTAMP_TZ 。

日期格式化元素中的大写字母

拼写出来的单词、缩写词或罗马数字中的大写字母在相应的格式元素中也跟着大写。例如，日期格式元素 DAY 产生的 MONDAY 也大写，Day 和 Monday 格式一样，day 和 monday 格式一样。

日期时间格式化中的标点符号和字符字面量

以下字符需要日期格式化，这些字符出现在返回值中的位置与格式化中字符的位置相同：

- 标点符号，例如连字符，斜杠，逗号，句号和冒号。
- 字符字面量，用双引号引起来。

OceanBase 数据库可以灵活的将字符串转换为日期。当您使用 TO_DATE 函数时，若输入字符串中的每个数字元素都包含格式化允许的最大位数，则格式字符串将与输入的字符串匹配。

示例 1：格式元素 **MM/YY**，其中 **02** 对应 **MM**，**07** 对应 **YY**。

执行以下语句：

```
SELECT TO_CHAR(TO_DATE('0207','MM/YY'),'MM/YY') FROM DUAL;
```

查询结果如下：

```
+----------------------------------------+
| TO_CHAR(TO_DATE('0207','MM/YY'),'MM/YY') |
+----------------------------------------+
| 02/07 |
+----------------------------------------+
```

示例 2：OceanBase 允许非字母数字字符与格式化中的标点字符匹配，**#** 对应 **/**。

执行以下语句：

```
SELECT TO_CHAR (TO_DATE('02#07','MM/YY'), 'MM/YY') FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------+
| TO_CHAR(TO_DATE('02#07','MM/YY'),'MM/YY') |
+-----------------------------------------+
| 02/07 |
+-----------------------------------------+
```

日期格式化元素与全球化支持

在 OceanBase 数据库中，日期时间格式化元素的语言可以通过参数 NLS_DATE_LANGUAGE 和 NLS_LANGUAGE 指定。默认值是 AMERICAN，不支持修改，所以不支持全球化。

示例：日期时间格式化的语言参数默认是 American，不支持其他语言。

```
SELECT TO_CHAR (SYSDATE, 'DD/MON/YYYY', 'nls_date_language=''Traditional Chinese'' ') FROM DUAL;
```

查询结果报错，语言参数不支持。

```
ERROR-12702: invalid NLS parameter string used in SQL function
```

更多信息

字符串到日期的转换规则

4.4.4 RR 日期时间格式化元素

RR 日期时间格式化元素类似于 YY 日期时间格式化元素，但它为跨世纪日期值存储提供了额外的灵活性。在 YY 日期时间格式化元素里，您需要指定年份的全部数字。而在 RR 日期时间格式化元素里，您只需指定年份数字的最后两位数，便可以存储日期值。

RR 日期时间格式化元素与 TO_DATE 函数一起使用，返回值的世纪根据指定的两位数字年份和当前年份的最后两位数字而变化。如果 YY 日期时间格式化元素与 TO_DATE 函数一起使用，则返回的年份始终与当前年份具有相同的前两位数字。

如果指定的两位数字年份是 00~49，当前年份的最后两位数字是 00~49，则返回的年份与当前年份具有相同的前两位数字。当前年份的后两位数字是 50~99，则返回年份的前两位数字比当前年份的前两位数字大 1。

如果指定的两位数字年份是 50~99，当前年份的后两位数字为 00 到 49，则返回年份的前两位数字比当前年份的前两位数字小 1。当前年份的最后两位数字是 50~99，则返回的年份与当前年份具有相同的前两位数字。

如下所示，RR日期时间格式化元素根据前两位数字不同的年份返回相同的值。假设这些查询是在 1950~1999 年期间发出的，执行以下语句：

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY')"Year1",
TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY')"Year2"FROM DUAL;
```

查询结果如下：

```
+-------+-------+
| Year1 | Year2 |
+-------+-------+
| 2017 | 1998 |
+-------+-------+
```

假设这些查询是在 2000~2049 年期间发出的，执行以下语句：

```
SELECT TO_CHAR(TO_DATE('27-OCT-98', 'DD-MON-RR'), 'YYYY')"Year1",
TO_CHAR(TO_DATE('27-OCT-17', 'DD-MON-RR'), 'YYYY')"Year2"FROM DUAL;
```

查询结果如下：

```
+-------+-------+
| Year1 | Year2 |
+-------+-------+
| 2017 | 1998 |
+-------+-------+
```

注意：无论是在 2000 年之前还是之后查询，都将返回相同的值。

4.4.5 字符串到日期的转换规则

将字符串值转换为日期值时，会有下列转换规则：

- 如果指定了数值格式化元素的所有数值（包括前导零点），则可以从日期字符串中省略格式化字符串中包含的标点符号。为两位数格式化元素（如 MM、DD 和 YY）指定 02 而不是 2。
- 您可以从日期字符串中省略在格式化字符串末尾找到的时间字段。
- 您可以在日期字符串中使用任何非字母数值字符来匹配格式化字符串中的标点符号。

4.5 空值

4.5.1 空值概述

空值（Null）指数据库表中无效的、未指定的、未知的或不可预知的值。空值的出现不受 NOT NULL 或 PRIMARY KEY 主键约束。任何包含 NULL 的算术表达式结果都为 NULL 。

OceanBase 支持以下 3 种空值类型。

SQL 函数中的空值

SQL 函数中的空值 指的是 SQL 函数的参数为空值，当 SQL 函数的参数为空值时，大多数标量函数都返回 NULL，分析函数会忽略空值。此类别中有以下两种 SQL 函数：

空值	说明
NVL 函数中的空值	在表达式中 NVL(expr1,expr2)中，如果 expr1 不是 NULL，返回 expr1，否则返回 expr2。
分析函数中的空值	使用 AVG，MAX，SUM，COUNT 等分析函数时，为 NULL 的纪录会被忽略。

比较条件中的空值

比较条件中的空值 指与任何其他条件做比较的 NULL 。测试空值只能用比较运算符 IS NULL 和 IS NOT NULL 。因为 NULL 表示缺少数据，所以 NULL 和其它值没有可比性，即不能用等于、不等于、大于或小于和其它数值比较，当然也包括空值本身。

条件判断表达式中的空值

条件判断表达式中的空值 指的是条件 = NULL、!= NULL、 NULL =、NULL ！= 中的 NULL，作逻辑判断使用，判断结果不返回任何行，即 UNKNOWN。

4.5.2 SQL 函数中的空值

SQL 函数中的空值指的是 SQL 函数的参数存在空值，当 SQL 函数的参数为空值时，大多数标量函数都返回 NULL，分析函数会忽略空值。您可以通过 NVL 函数的返回值确定空值。

NVL 函数中的空值

NVL 函数的表达式为 NVL(expr1,expr2)，如果 expr1 不是 NULL，返回 expr1，否则返回 expr2。

如下所示，给定 expr1 参数为NULL，查询 NVL(expr1，0) 表达式的返回值。执行以下语句：

```
SELECT NVL(NULL,0) FROM DUAL;
```

查询结果如下：

```
+-------------+
| NVL(NULL,0) |
+-------------+
| 0 |
+-------------+
```

若expr1 是 NULL，则表达式 NVL(expr1，0) 的返回值为 0；若expr1不是 NULL，则表达式的返回值为NULL。

分析函数中的空值

在使用AVG、MAX、SUM 或 COUNT等分析函数时，为 NULL 的纪录会被忽略。

如下所示，向 **tbl_a** 表中插入数据并执行以下语句：

```
CREATE TABLE tbl_a (col_a varchar2(1), col_b int);
INSERT INTO tbl_a VALUES (NULL, 3);
INSERT INTO tbl_a VALUES (NULL, NULL);
INSERT INTO tbl_a VALUES (NULL, 1);
```

执行以下语句：

```
SELECT * FROM tb1_a;
```

查询结果如下：

```
+-------+-------+
| COL_A | COL_B |
+-------+-------+
| NULL | 3 |
+-------+-------+
| NULL | NULL |
+-------+-------+
| NULL | 1 |
+-------+-------+
```

查询的结果如下：

```
SELECT AVG(col_b) FROM tbl_a; -- 结果为 2 ，
SELECT MAX(col_b) FROM tbl_a; -- 结果为 3
SELECT SUM(col_b) FROM tbl_a; -- 结果为 4
SELECT COUNT(col_b) FROM tbl_a; -- 结果为 2
SELECT COUNT(col_a) FROM tbl_a; -- 结果为 0
SELECT COUNT(*) FROM tbl_a; -- 结果为 3
```

NULL 的纪录被忽略了。

4.5.3 条件判断表达式中的空值

条件判断表达式中的空值指的是条件 = NULL、!= NULL、 NULL =、NULL ！= 中的 NULL，作逻辑判断使用，判断结果不返回任何行，即 UNKNOWN。

在 OceanBase 中，测试空值要用比较运算符 IS NULL ，返回结果为 TRUE 或 FALSE 。但是条件判断表达式中空值的判断结果 UNKNOWN 与 FALSE 不同，NOT FALSE 判断结果为 TRUE， NOT UNKNOWN 判断结果仍然为 UNKNOWN。

如下所示，根据 A 值判断条件判断表达式的结果。

条件	A 值	结果
A = NULL	10	UNKNOWN
A != NULL	10	UNKNOWN
A = NULL	NULL	UNKNOWN
A != NULL	NULL	UNKNOWN
A = 10	NULL	UNKNOWN
A != 10	NULL	UNKNOWN

如果在 SELECT 语句的 WHERE 子句中使用了判断结果为 UNKNOWN 的条件，则该查询将不返回任何行。

4.5.4 比较条件中的空值

比较条件中的空值指与任何其他条件做比较的 NULL 。测试空值只能用比较运算符 IS NULL 和 IS NOT NULL 。因

为 NULL 表示缺少数据，所以 NULL 和其它值没有可比性，即不能用等于、不等于、大于或小于和其它数值比较，当然也包括空值本身。

另外，OceanBase 在计算 DECODE 函数时认为两个空值是相等的。若两个空值出现在复合键中，则它们也相等。

如下所示，根据 A 值判断比较条件的结果。

条件	A 值	结果
A IS NULL	10	FALSE
A IS NOT NULL	10	TRUE
A IS NULL	NULL	TRUE
A IS NOT NULL	NULL	FALSE

4.6 注释

4.6.1 注释概述

OceanBase 中用户可以创建三种注释：

- SQL 语句的注释 ：被存储为执行 SQL 语句的应用程序代码的一部分。
- Schema 和非 Schema 对象的注释 ：与对象本身的元数据一起存储在数据字典中。
- Hint ：一种在SQL 语句中将指令传递给 OceanBase 数据库优化器或服务器的一种注释。

4.6.2 SQL 语句的注释

注释可以使应用程序更易于阅读和维护。例如，您可以在语句中用注释以描述该语句在应用程序中的用途。除 Hint 外，SQL 语句中的注释不会影响语句的执行。

注释可以出现在语句中的任何关键字、参数或标点符号之间。您可以通过两种方式在语句中添加注释：

- 以斜杠和星号（/*）为开头的注释。斜杠和星号后跟着注释的文本。此文本可以跨越多行，并用星号和斜杠（*/）结束注释。开头和结尾的符号不必与文本用空格或换行符进行分隔。
- 以两个连字符（--）为开头的注释。符号后跟着注释的文本。此文本不能扩展到新行，并以换行符结束注释。

一些用于输入 SQL 的工具有附加的限制。例如，如果使用的是 SQL Plus，在默认情况下，多行注释中不能有空行。

一个 SQL 语句可以同时包含这两种风格的多个注释。注释的文本可以包含数据库字符集中的任何可打印字符。

以下示例展示了多种形式的以斜杠和星号（/*）为开头的注释：

```
SELECT last_name, employee_id, salary + NVL(commission_pct, 0),
job_id, e.department_id
/* Select all employees whose compensation is
```

```
greater than that of Pataballa.*/
FROM employees e, departments d
/*The DEPARTMENTS table is used to get the department name.*/
WHERE e.department_id = d.department_id
AND salary + NVL(commission_pct,0) > /* Subquery: */
(SELECT salary + NVL(commission_pct,0)
/* total compensation is salary + commission_pct */
FROM employees
WHERE last_name = 'Pataballa')
ORDER BY last_name, employee_id;
```

以下示例中的语句包含多种形式的以两个连字符（- -）为开头的注释：

```
SELECT last_name, -- select the name
employee_id -- employee id
salary + NVL(commission_pct, 0), -- total compensation
job_id, -- job
e.department_id -- and department
FROM employees e, -- of all employees
departments d
WHERE e.department_id = d.department_id
AND salary + NVL(commission_pct, 0) > -- whose compensation
-- is greater than
(SELECT salary + NVL(commission_pct,0) -- the compensation
FROM employees
WHERE last_name = 'Pataballa') -- of Pataballa
ORDER BY last_name -- and order by last name
employee_id -- and employee id.
;
```

4.6.3 Schema 与非 Schema 对象的注释

可以使用 COMMENT 语句将注释与 Schema 对象（表、视图、物化视图、运算符、索引类型）或非 Schema 对象（Edition）关联起来。还可以在表模式对象的列上创建注释。与 Schema 和非 Schema 对象关联的注释存储在数据字典中。

4.6.4 Hint 说明

Hint 是 SQL 语句中将指令传递给 OceanBase 数据库优化器或服务器的一种注释。通过 Hint 可以使优化器或服务器生成某种特定的计划。 一般情况下，优化器会为用户查询选择最佳的执行计划，不需要用户主动使用 Hint 指定，但在某些场景下，优化器生成的执行计划可能不满足用户的要求，这时就需要用户使用 Hint 来主动指定并生成特殊的执行计划。

Hint 应该少用，仅在您收集了相关表的统计信息并且使用 EXPLAIN PLAN 语句在没有 Hint 的情况下评估了优化器计划之后，才谨慎考虑使用。更改数据库条件以及在后续版本中增强查询性能可能会导致您代码中的 Hint 对性能产生重大影响。

Hint 有以下几类：

- 与连接顺序相关的 Hint

- 与联接操作相关的 Hint
- 与并行执行相关的 Hint
- 与访问路径相关的 Hint
- 与查询装换相关的 Hint
- 与查询策略相关的 Hint
- 其他 Hint

Hint 的使用

一个语句块只能有一个注释包含 Hint，并且该注释必须跟随 SELECT、UPDATE、INSERT、MERGE 或 DELETE 关键字。

以下是 Hint 在语句块注释中的语法格式：

```
/*+[hint text]*/
```

Hint 从语法上看是一种特殊的 SQL 注释, 所不同的是在注释的左标记后增加了一个加号（+）。 如果服务器端无法识别 SQL 语句中的 Hint，那么优化器会选择忽略用户指定的 Hint 而使用默认计划所生成逻辑。另外需要指明的是，Hint 只影响优化器所生成的计划的逻辑，而不影响 SQL 语句的语义。

以下是定义 Hint 时需要注意的一些规则：

- 加号（+）使数据库将注释解释为 Hint 列表。加号必须紧跟在注释左标记符后，不允许有空格。
- 加号（+）和 Hint 文本之间的空格是可选的。如果注释中包含多个 Hint，则 Hint 间至少用一个空格进行分隔。
- Hint 包含拼写错误或语法错误时会被忽略。但是，数据库会考虑在同一注释中其他正确指定的 Hint。
- 不跟随 DELETE、INSERT、MERGE、SELECT 或 UPDATE 关键字的 Hint 无效。
- Hint 的组合相互冲突时 Hint 无效。但是，数据库会在同一注释中考虑其他不冲突的 Hint。
- 数据库环境使用 PL/SQL 版本 1 时 Hint 无效，例如 Forms 版本 3 触发器。

在 Hint 中定义查询块

您可以通过在许多 Hint 中定义一个可选的查询块名称，以此来指定该 Hint 适用的查询块。使用此语法，允许您在外部查询中指定一个应用于嵌入式视图的 Hint。

查询块参数的语法格式为 @queryblock，其中 queryblock 是在查询中被指定的查询块的标识符。queryblock 标识符可以是系统生成的，也可以是用户自己指定的。当您在查询块中直接指定要应用的 Hint 时，将忽略 @queryblock。

- 系统生成的标识符可以通过对查询使用 EXPLAIN PLAN 生成，预转换查询块的名称可以通过对使用了 NO_QUERY_TRANSFORMATION Hint 的查询运行 EXPLAIN PLAN 生成。
- 可以使用 QB_NAME 来指定用户自定义的名称。

定义全局 Hint

许多 Hint 既可以应用于特定的表或索引，也可以更全局地应用于视图中的表或一部分索引的列。语法元素 tablespec 和 indexspec 定义了这些全局 Hint。

以下是 tablespec 的语法：

```
[ view.[ view. ]... ]table
```

您必须完全按照在语句中显示的方式来指定要访问的表。如果该语句使用表的别名，则在 Hint 中使用也使用别名而不是表名。但是，即使 Schema 名称出现在语句中，也不要在 Hint 中使用的表名中包含 Schema 名称。

注意：使用 tablespec 子句指定全局 Hint 对于使用 ANSI 连接的查询不起作用，因为优化器在解析期间会生成额外的视图。相反，可以通过 @queryblock 来指定该提示所应用到的查询块。

以下是 indexspec 的语法：

```
{ index | ([table. ]column [ [ table. ]column ]...)}
```

在 Hint 的说明部分，当 tablespec 后跟着 indexspec 时，允许但不要求使用逗号来分隔表名和索引名。也允许（但不是必需）使用逗号分隔多次出现的 indexspec。

4.6.5 与联接顺序相关的 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 和 在 Hint 中定义查询块 小节。

LEADING Hint

LEADING Hint 指示优化器在执行计划中使用指定的表集作为前缀，它可以用来指定表的联接顺序。这个 Hint 比 ORDERED Hint 更通用。

以下是 LEADING Hint 的语法：

```
/*+ LEADING ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

LEADING Hint 为确保按照用户指定的顺序联接表，所以会进行严格的检查。如果发现 Hint 指定的 table_name 不存在，则 LEADING Hint 失效。如果发现 Hint 中存在重复表，则 LEADING Hint 失效。如果在优化器联接期间，无法找到对应的表，那么该表及后面的表指定的联接顺序失效，该表前面指定的顺序依然有效。如果由于联接图中的依赖关系，无法首先按照指定的顺序联接指定的表，则 LEADING Hint 失效。如果指定两个或多个相互冲突的 LEADING Hint，则 LEADING Hint 失效。如果您指定了 ORDERED Hint，它将覆盖所有的 LEADING Hint。

示例如下：

```
SELECT /*+ LEADING(e j) */ *
FROM employees e, departments d, job_history j
```

```
WHERE e.department_id = d.department_id
AND e.hire_date = j.start_date;
```

ORDERED Hint

ORDERED Hint 指示数据库按照表在 FROM 子句中出现的顺序联接表。建议使用 LEADING Hint，它比 ORDERED Hint 更通用。

以下是 ORDERED Hint 的语法：

```
/*+ ORDERED */
```

当您从需要联接的SQL语句中省略 ORDERED Hint 时，将由优化器将选择联接表的顺序。但是优化器不知道从每个表中要选择的行数，此时您可以使用 ORDERED Hint 来指定联接顺序。这样使您能够比优化器更好地选择内部表和外部表。如果在指定该 ORDERED Hint 后发生了改写，那么就按照改写后的语句中的 FROM 子句的顺序联接表。

示例如下：

```
SELECT /*+ ORDERED */ o.order_id, c.customer_id, l.unit_price * l.quantity
FROM customers c, order_items l, orders o
WHERE c.cust_last_name = 'Taylor'
AND o.customer_id = c.customer_id
AND o.order_id = l.order_id;
```

4.6.6 与联接操作相关的 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 和 定义全局 Hint 小节。

USE_MERGE Hint

USE_MERGE Hint 指示优化器使用一个 sort-merge 联接将每个指定的表与另一个行资源联接起来。建议在使用 USE_NL 和 USE_MERG Hint 时和 LEADING 与 ORDERED Hint 一起使用。当被引用的表是联接的内部表时，优化器将使用这些提示。如果被引用的表是外部表，则忽略 Hint。

以下是 USE_MERGE Hint 的语法：

```
/*+ USE_MERGE ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

USE_MERGE 指定表作为内部表时候使用 MERGE-JOIN 算法。OceanBase 中使用 MERGE-JOIN 算法时必须有等值条件的 join-condition，因此无等值条件的两个表做联结时，USE_MERGE 失效。

以下是 USE_MERGE 的示例：

```
SELECT /*+ USE_MERGE(employees departments) */ *
FROM employees, departments
```

```
WHERE employees.department_id = departments.department_id;
```

NO_USE_MERGE Hint

NO_USE_MERGE Hint 指示优化器在使用指定表作为内部表并联接到另一个行资源时排除 USE_MERGE Hint 使用的联接。

以下是 NO_USE_MERGE Hint 的语法：

```
/*+ NO_USE_MERGE ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

以下是 NO_USE_MERGE Hint 的示例：

```
SELECT /*+ NO_USE_MERGE(e d) */ *
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

USE_HASH Hint

USE_HASH Hint 指示优化器使用 HASH-JOIN 算法将每个指定的表与另一个行资源联接起来。

以下是 USE_HASH Hint 的语法：

```
/*+ USE_HASH ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

以下是 USE_HASH Hint 的示例：

```
SELECT /*+ USE_HASH(l h) */ *
FROM orders h, order_items l
WHERE l.order_id = h.order_id
AND l.order_id > 2400;
```

NO_USE_HASH Hint

NO_USE_HASH Hint 指示优化器在使用指定表作为内部表并联接到另一个行资源时排除 USE_HASH Hint 使用的联接。

以下是 NO_USE_HASH Hint 的语法：

```
/*+ NO_USE_HASH ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

以下是 NO_USE_HASH Hint 的示例：

```
SELECT /*+ NO_USE_HASH(e d) */ *
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

USE_NL Hint

USE_NL Hint 指示优化器使用嵌套循环联接将每个指定的表连接到另一个行资源，并使用指定的表作为内部表，指定表作为内部表时使用 NL-JOIN 算法。建议在使用 USE_NL 和 USE_MERG Hint 时和 LEADING 与 ORDERED Hint 一起使用。当被引用的表是联接的内部表时，优化器将使用这些提示。如果被引用的表是外部表，则忽略 Hint。

以下是 USE_NL Hint 的语法：

```
/*+ USE_NL ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

在下面的示例中，Hint 强制执行了嵌套循环，通过全表扫描访问了 **orders** 并且筛选条件 l.order_id = h.order_id 应用在了每一行。对于满足筛选条件的每一行，通过索引 **order_id** 访问 **order_items**：

```
SELECT /*+ USE_NL(l h) */ h.customer_id, l.unit_price * l.quantity
FROM orders h, order_items l
WHERE l.order_id = h.order_id;
```

NO_USE_NL Hint

NO_USE_NL Hint 指示优化器在使用指定表作为内部表并联接到另一个行资源时排除 Nest-Loop 联接。

以下是 NO_USE_NL Hint 的语法：

```
/*+ NO_USE_NL ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

以下是 NO_USE_NL Hint 的示例：

```
SELECT /*+ NO_USE_NL(e d) */ *
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

USE_BNL Hint

USE_BNL Hint 指示优化器使用块嵌套循环联接将每个指定的表连接到另一个行资源，并使用指定的表作为内部表，指定表作为内部表时使用 BNL-JOIN 算法。建议在使用 USE_BNL Hint 时和 LEADING 与 ORDERED Hint 一起使用。当被引用的表是联接的内部表时，优化器将使用这些提示。如果被引用的表是外部表，则忽略 Hint。

以下是 USE_BNL Hint 的语法：

```
/*+ USE_BNL ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

在下面的示例中，Hint 强制执行了块嵌套循环，通过全表扫描访问了 **orders** 并且筛选条件 l.order_id = h.order_id 应用在了每一行。对于满足筛选条件的每一行，通过索引 **order_id** 访问 **order_items**：

```
SELECT /*+ USE_BNL(l h) */ h.customer_id, l.unit_price * l.quantity
```

```
FROM orders h, order_items l
WHERE l.order_id = h.order_id;
```

NO_USE_BNL Hint

NO_USE_BNL Hint 指示优化器在使用指定表作为内部表并联接到另一个行资源时排除 USE_BNL Hint 使用的联接。

以下是 NO_USE_BNL Hint 的语法：

```
/*+ NO_USE_BNL ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

以下是 NO_USE_BNL Hint 的示例：

```
SELECT /*+ NO_USE_BNL(e d) */ *
FROM employees e, departments d
WHERE e.department_id = d.department_id;
```

4.6.7 与并行执行相关的 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 和 定义全局 Hint 小节。

PARALLEL Hint

PARALLEL Hint 是语句级的 Hint，用来指示优化器指定并行操作可使用的并行服务器的数量。此 Hint 将覆盖初始化参数 PARALLEL_DEGREE_ POLICY 的值。该 Hint 适用于语句的 SELECT、INSERT、MERGE、UPDATE 和 DELETE 部分，以及表扫描的部分。如果违反了任何的并行限制，则 PARALLEL Hint 被忽略。

以下是 PARALLEL Hint 的语法：

```
/*+ PARALLEL [ ( DEFAULT | AUTO | MANUAL | integer ) ] */
```

注意：如果还进行了排序或分组操作，那么可以使用的服务器数量是 PARALLEL Hint 中的值的两倍。

PARALLEL Hint 中指定参数的值时：

- PARALLEL：数据库计算并行度，可以为 2 或更大。语句一直并行执行。
- PARALLEL(DEFAULT) ：优化器计算的并行度等于所有参与实例上可用的 CPU 数量乘以初始化参数 PARALLEL_THREADS_PER_CPU 的值。
- PARALLEL(AUTO) ：数据库计算并行度，结果可以大于等于 1，如果计算出的并行度为 1，则该语句按顺序运行。
- PARALLEL(MANUAL)：优化器被强制使用语句中设置的对象的并行度。
- PARALLEL(integer)：优化器使用参数 integer 指定的整数值为并行度。

以下示例中数据库计算并行度，并且语句一直并行执行：

```
SELECT /*+ PARALLEL */ last_name
FROM employees;
```

以下示例中数据库计算并行度，但是并行度为 1，所以该语句串行执行：

```
SELECT /*+ PARALLEL (AUTO) */ last_name
FROM employees;
```

在以下示例中，PARALLEL Hint 建议优化器使用语句中设定的，当前对表本身有效的并行度 5：

```
CREATE TABLE parallel_table (col1 number, col2 VARCHAR2(10)) PARALLEL 5;
SELECT /*+ PARALLEL (MANUAL) */ col2
FROM parallel_table;
```

USE_PX Hint

USE_PX Hint 强制指示服务器在执行 SQL 语句时使用 PX 模式，PX 模式允许在执行语句时采用多线程方式。一般 USE_PX Hint 和 PARALLEL Hint 配合使用。

以下是 USE_PX Hint 的语法：

```
/*+ USE_PX */
```

示例如下：

```
SELECT /*+ USE_PX PARALLEL(4)*/ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

NO_USE_PX Hint

NO_USE_PX Hint 强制指示服务器在执行 SQL 语句时避免使用 PX 模式。

以下是 NO_USE_PX Hint 的语法：

```
/*+ NO_USE_PX */
```

示例如下：

```
SELECT /*+ NO_USE_PX*/ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

PQ_DISTRIBUTE Hint

PQ_DISTRIBUTE Hint 指示优化器怎样在程序（查询）服务器和消耗（负载）查询服务器之间分配行。您可以通过该 Hint 控制联接或负载的行分布。

以下是 PQ_DISTRIBUTE Hint 的语法：

```
/*+ PQ_DISTRIBUTE
( [ @ queryblock ] tablespec
{ distribution | outer_distribution inner_distribution }
) */
```

控制负载的分配

您可以控制并行语句 INSERT ... SELECT 和并行语句 CREATE TABLE ... AS SELECT 的行分布，以此来确定如何在程序（查询）服务器和消耗（负载）服务器之间进行行分配。使用语法的上分支来指定分发方法。分布方法的值及其语义如下表所示：

分布方法	说明
NONE	没有分配。即将查询和负载操作组合到每个查询服务器中。所有服务器将加载所有分区。这种分配方法的缺失有助于避免在没有偏离的情况下行分配的开销。由于空段或语句中的谓词会过滤掉查询评估的所有行，因此可能会发生偏离。如果由于使用此方法而发生偏斜，则请改用 RANDOM 或 RANDOM_ LOCAL 分布。**注意**：请谨慎使用此分配方法。每个进程加载的 PGA 内存最少需要 512 KB。如果还使用压缩，则每台服务器消耗大约 1.5 MB 的 PGA 内存
PARTITION	此方法使用 tablespec 的分区信息将行从查询服务器分发到消耗服务器。当不可能或不希望将查询和加载操作组合在一起时和当正在加载的分区数量大于或等于加载服务器的数量并且输入数据将均匀地分布在正在加载的分区之间（即没有偏离）时，请使用此分步方法。
RANDOM	此方法以循环方式将来自程序的行分发给消耗。当输入数据高度倾斜时，使用这种分布方法。
RANDOM_LOCAL	此方法将来自程序的行分布到一组服务器，这些服务器负责维护给定的一组服务器。两个或多个服务器可以加载同一分区，但是没有服务器加载所有分区。当输入数据发生偏移并且由于内存限制而无法合并查询和加载操作时，请使用此分布方法。

例如，在以下直接装入插入操作中，该操作的查询和负载部分被组合到每个查询服务器中：

```
INSERT /*+ APPEND PARALLEL(target_table, 16) PQ_DISTRIBUTE(target_table, NONE) */
INTO target_table
SELECT * FROM source_table;
```

在下面的示例中，创建表时优化器使用表 **target_table** 的分区来分配行：

```
CREATE /*+ PQ_DISTRIBUTE(target_table, PARTITION) */ TABLE target_table
NOLOGGING PARALLEL 16
PARTITION BY HASH (l_orderkey) PARTITIONS 512
AS SELECT * FROM source_table;
```

控制联接的分配

您可以通过指定两种分配方法来控制联接的分配方法，如语法中的下部分支所示，一种外部表的分布方法，一种内部表的分布方法：

- outside_distribution 是外部表的分布方法。

- inner_distribution 是内部表的分布方法。

分布方法的值是 HASH、BROADCAST、PARTITION 和 NONE。只有下表中的 6 种分布方法组合是有效的：

分布方法	说明
HASH, HASH	使用联接键上的哈希函数，将每个表的行映射到消耗查询服务器。映射完成后，每个查询服务器都会在一对结果分区之间执行联接。当表的大小可比较并且联接操作是通过哈希联接或排序合并联接实现时，建议使用此分布方法。
BROADCAST, NONE	外部表的所有行都广播到每个程序查询服务器。内部表行是随机分区的。当外部表与内部表相比非常小时，建议使用此分布方法。通常，当内部表大小乘以查询服务器的数量大于外部表大小时，也建议使用此分布方法。
NONE, BROADCAST	内部表的所有行都广播给每个消耗查询服务器。外部表行是随机分区的。当内部表与外部表相比非常小时，建议使用此分布方法。通常，当内部表大小乘以查询服务器的数量小于外部表大小时，也建议使用此分布方法。
PARTITION, NONE	外部表的行使用以内部表的分区进行映射。内部表必须在联接键上进行分区。当外部表的分区数等于或几乎等于查询服务器数的倍数时，建议使用此分布方法。例如，有 14 个分区和 15 个查询服务器。**注意**：如果内部表未分区或未在分区键上等分联接时，则优化器将忽略此 Hint。
NONE, PARTITION	内部表的行使用外部表的分区进行映射。外部表必须在联接键上进行分区。当外部表的分区数等于或几乎等于查询服务器数的倍数时，建议使用此分布方法。例如，有 14 个分区和 15 个查询服务器。**注意**：如果外部表未在分区键上进行分区或未等分联时接，则优化器将忽略此 Hint。
NONE, NONE	每个查询服务器在一对匹配的分区之间执行联接操作，每个表中都有一个。两个表必须在连接键上等分。

例如，给定两个使用哈希联接来联接表 **r** 和 **s**，以下查询包含使用哈希分配的 Hint：

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s HASH, HASH) USE_HASH (s)*/ column_list
FROM r,s
WHERE r.c=s.c;
```

要广播外部表 **r**，查询语句为：

```
SELECT /*+ORDERED PQ_DISTRIBUTE(s BROADCAST, NONE) USE_HASH (s) */ column_list
FROM r,s
WHERE r.c=s.c;
```

4.6.8 与访问路径相关的 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 和 定义全局 Hint 小节。

INDEX Hint

INDEX Hint 指示优化器对指定的表使用索引扫描。您可以将INDEX Hint 用于基于函数、域、B - 树、位图和位图连接的索引。

以下是 INDEX Hint 的语法：

```
/*+ INDEX ( [ @ queryblock ] tablespec [ indexspec [ indexspec ]... ] ) */
```

Hint 的行为取决于 indexspec 规范：

- 如果 INDEX Hint 指定了一个单个可用索引，则数据库将对该索引执行扫描。优化器不考虑全表扫描或表上另一个索引的扫描。
- 如果 INDEX Hint 指定了可用索引的列表，那么优化器将考虑扫描列表中每个索引的成本，然后以最低的成本执行索引扫描。如果数据库从这个列表中扫描多个索引并合的访问路径成本最低，数据库将选用这种扫描方案。数据库不考虑对没有在列表中的索引进行全表扫描或扫描。
- 如果 INDEX Hint 没有指定具体的索引，那么优化器将考虑表上每个可用索引的扫描成本，然后以最低的成本执行索引扫描。如果数据库扫描多个索引并合的访问路径成本最低，数据库将选用这种扫描方案。优化器不考虑全表扫描。

示例如下：

```
SELECT /*+ INDEX (employees emp_department_ix)*/ employee_id, department_id
FROM employees
WHERE department_id > 50;
```

FULL Hint

FULL Hint 指示优化器对指定的表执行全表扫描。

以下是 FULL Hint 的语法：

```
/*+ FULL ( [ @ queryblock ] tablespec ) */
```

示例如下：

```
SELECT /*+ FULL(e) */ employee_id, last_name
FROM hr.employees e
WHERE last_name LIKE :b1;
```

数据库对表 **employees** 执行一次完整的表扫描来执行这条语句，即使有一个由 WHERE 子句中的条件提供的索引在列 **last_name** 上。

在 FROM 子句中，表 **employees** 有个别名 **e**，因此 Hint 必须根据表的别名而不是名称来引用该表。即使在 FROM 子句中指定了 Schema 名，也不要在 Hint 中引用它们。

4.6.9 与查询转换相关的 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档

Hint 说明 中的 在 Hint 中定义查询块 和 定义全局 Hint 小节。

NO_REWRITE Hint

NO_REWRITE Hint 指示优化器为禁用查询去重写查询块，并覆盖了参数 QUERY_REWRITE_ENABLED 的设置。
以下是 NO_REWRITE 的语法：

```
/*+ NO_REWRITE [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+ NO_REWRITE */ sum(s.amount_sold) AS dollars
FROM sales s, times t
WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

NO_EXPAND Hint

NO_EXPAND Hint 指示优化器不要对 WHERE 子句中具有 OR 条件或 IN 列表的查询考虑 OR 扩展。 通常，优化器会使用 OR 扩展，当确定使用 OR 扩展的成本低于不使用它时。

以下是 NO_EXPAND Hint 的语法：

```
/*+ NO_EXPAND [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+ NO_EXPAND */ *
FROM employees e, departments d
WHERE e.manager_id = 108
OR d.department_id = 110;
```

USE_CONCAT Hint

USE_CONCAT Hint 指示优化器使用 UNION ALL 运算符将查询 WHERE子句中的组合 OR 条件转换为复合查询。
如果没有这个 Hint，则仅当使用串联查询的成本低于比没有串联查询的成本时，才会发生此转换。
USE_CONCAT Hint 将覆盖成本注意事项。

以下是 USE_CONCAT Hint 的语法：

```
/*+ USE_CONCAT [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+ USE_CONCAT */ *
FROM employees e
WHERE manager_id = 108
```

```
OR department_id = 110;
```

MERGE Hint

MERGE Hint 使您可以在查询中合并视图。

以下是 MERGE Hint 的语法：

```
/*+ MERGE [ ( @ queryblock ) | ( [ @ queryblock ] tablespec ) ] */
```

如果视图的查询块在 SELECT 列表中包含 GROUP BY 子句或 DISTINCT 运算符时，只有启用了复杂的视图合并后，优化器才能将视图合并到正在访问的语句中。 如果子查询不相关，则也可以使用复杂合并将 IN 子查询合并到访问语句中。

示例如下：

```
SELECT /*+ MERGE(v) */ e1.last_name, e1.salary, v.avg_salary
FROM employees e1,
(SELECT department_id, avg(salary) avg_salary
FROM employees e2
GROUP BY department_id) v
WHERE e1.department_id = v.department_id
AND e1.salary > v.avg_salary
ORDER BY e1.last_name;
```

当不带参数使用 MERGE Hint 时，应将其放在视图查询块中。 当视图名称作为参数使用 MERGE Hint 时，应将其放在周边查询中。

NO_MERGE Hint

NO_MERGE Hint 指示优化器不要将外部查询和任何内联视图查询合并到单个查询中。

以下是 NO_MERGE Hint 的语法：

```
/*+ NO_MERGE [ ( @ queryblock ) | ( [ @ queryblock ] tablespec ) ] */
```

此 Hint 会影响您访问视图的方式。 例如，以下语句导致视图 **seattle_dept** 不被合并：

```
SELECT /*+ NO_MERGE(seattle_dept) */ e1.last_name, seattle_dept.department_name
FROM employees e1,
(SELECT location_id, department_id, department_name
FROM departments
WHERE location_id = 1700) seattle_dept
WHERE e1.department_id = seattle_dept.department_id;
```

在视图查询块中使用 NO_MERGE Hint 时，不需要为它指定参数。 在周围的查询中使用 NO_MERGE Hint 时，需要将视图名称指定为它的参数。

UNNEST Hint

UNNEST Hint 指示优化器不要嵌套而是并将子查询的主体合并到包含该 Hint 的查询块的主体中，从而使优化器在评估访问路径和联接时将它们一起考虑在内。

以下是 UNNEST Hint 的语法：

```
/*+ UNNEST [ ( @ queryblock ) ] */
```

在取消嵌套子查询之前，优化器首先验证子查询是否有效，并且必须通过探索和查询优化测试。 使用 UNNEST Hint 时优化器仅检查子查询块的有效性。 如果子查询块有效，那么将直接取消嵌套子查询，而无需通过探索和查询优化测试。

示例如下：

```
SELECT AVG(t1.c) FROM t1
WHERE t1.b >=
(SELECT /*+unnest*/AVG(t2.b)
FROM t2
WHERE t1.a = t2.a);
```

NO_UNNEST Hint

NO_UNNEST Hint 用来关闭取消嵌套的操作。

以下是 NO_UNNEST Hint 的语法：

```
/*+ NO_UNNEST [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+no_unnest(@qb1)*/AVG(t1.c)
FROM t1 WHERE t1.b >=
(SELECT /*+qb_name(qb1)*/AVG(t2.b)
FROM t2 )
WHERE t1.a = t2.a);
```

PLACE_GROUP_BY Hint

PLACE_GROUP_BY Hint指示优化器采用 GROUP BY 位置替换规则，此时优化器不考虑转换之后的代价增大。

以下是PLACE_GROUP_BY Hint的语法：

```
/*+ PLACE_GROUP_BY [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+place_group_by*/SUM(t1.c),SUM(t2.c) FROM t1, t2
WHERE t1.a = t2.a AND t1.b > 10 AND t2.b > 10
```

```
GROUP BY t1.a;
```

NO_PLACE_GROUP_BY Hint

NO_PLACE_GROUP_BY Hint 用来关闭 GROUP BY 位置替换转换。

以下是NO_PLACE_GROUP_BY Hint的语法：

```
/*+ NO_PLACE_GROUP_BY [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+no_place_group_by*/SUM(t1.c),SUM(t2.c) FROM t1, t2
WHERE t1.a = t2.a AND t1.b > 10 AND t2.b > 10
GROUP BY t1.a;
```

NO_PRED_DEDUCE Hint

NO_PRED_DEDUCE Hint 用来指示优化器不是用谓词推导转换规则。

以下是NO_PRED_DEDUCE Hint的语法：

```
/*+ NO_PRED_DEDUCE [ ( @ queryblock ) ] */
```

示例如下：

```
SELECT /*+no_pred_deduce(@qb1)*/ *
FROM (
SELECT /*+no_merge qb_name(qb1)*/ t1.a, t2.b
FROM t1, t2
WHERE t1.a = t2.a) v, t3
WHERE t3.a = 1 AND t3.a = v.a;
```

4.6.10 与查询策略相关的 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 和 定义全局 Hint 小节。

USE_JIT Hint

USE_JIT Hint 指示服务器在执行 SQL 语句时强制使用 JIT 模式编译执行表达式。

以下是 USE_JIT Hint 的语法：

```
/*+ USE_JIT */
```

示例如下：

```
SELECT /*+ USE_JIT*/ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

NO_USE_JIT Hint

NO_USE_JIT Hint 指示服务器在执行 SQL 语句时避免使用 JIT 模式编译执行表达式。

以下是 NO_USE_JIT Hint 的语法：

```
/*+ NO_USE_JIT*/
```

示例如下：

```
SELECT /*+NO_USE_JIT*/ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

USE_HASH_AGGREGATION Hint

USE_HASH_AGGREGATION Hint 指示优化器在生成计划时强制使用 HASH 聚合算法运行该 SQL 语句。

以下是 USE_HASH_AGGREGATION Hint 的语法：

```
/*+ USE_HASH_AGGREGATION */
```

示例如下：

```
SELECT /*+ USE_HASH_AGGREGATION */ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

NO_USE_HASH_AGGREGATION Hint

NO_USE_HASH_AGGREGATION Hint 指示优化器在执行 SQL 语句时避免使用 HASH 聚合算法运行该语句。

以下是 NO_USE_HASH_AGGREGATION Hint 的语法：

```
/*+ NO_USE_HASH_AGGREGATION */
```

示例如下：

```
SELECT /*+ NO_USE_HASH_AGGREGATION */ e.department_id, sum(e.salary)
```

```
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

USE_LATE_MATERIALIZATION Hint

USE_LATE_MATERIALIZATION Hint 指示优化器延迟物化视图。

以下是 USE_LATE_MATERIALIZATION Hint 的语法：

```
/*+ USE_LATE_MATERIALIZATION */
```

示例如下：

```
SELECT /*+ USE_LATE_MATERIALIZATION*/ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

NO_USE_LATE_MATERIALIZATION Hint

NO_USE_LATE_MATERIALIZATION Hint 指示优化器禁止延迟物化视图。

以下是 NO_USE_LATE_MATERIALIZATION Hint 的语法：

```
/*+ NO_USE_LATE_MATERIALIZATION */
```

示例如下：

```
SELECT /*+ NO_USE_LATE_MATERIALIZATION*/ e.department_id, sum(e.salary)
FROM employees e
WHERE e.department_id = 1001;
GROUP BY e.department_id;
```

USE_NL_MATERIALIZATION Hint

USE_NL_MATERIALIZATION Hint 强制指示优化器指定表为内部表（子树）时生成一个物化算子来缓存数据。

以下是 USE_NL_MATERIALIZATION Hint 的语法：

```
/*+ USE_NL_MATERIALIZATION ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

示例如下：

```
SELECT /*+ USE_NL_MATERIALIZATION(departments) */ *
FROM employees, departments
WHERE employees.department_id = departments.department_id;
```

NO_USE_NL_MATERIALIZATION Hint

NO_USE_NL_MATERIALIZATION Hint 强制指示优化器在指定表为内部表（子树）时避免生成一个物化算子来缓存数据。

以下是 NO_USE_NL_MATERIALIZATION Hint 的语法：

```
/*+ NO_USE_NL_MATERIALIZATION ( [ @ queryblock ] tablespec [ tablespec ]... ) */
```

示例如下：

```
SELECT /*+ NO_USE_NL_MATERIALIZATION(departments) */ *
FROM employees, departments
WHERE employees.department_id = departments.department_id;
```

4.6.11 其他 Hint

注意：关于本篇文档所有语法中引用的 queryblock、tablespec 和 indexspec 等元素的更多信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 和 定义全局 Hint 小节。

QB_NAME Hint

使用 QB_NAME Hint 来定义查询块的名称。然后，可以在外部查询的 Hint 中使用这个名称，也可以在内联视图的 Hint 中使用这个名称，从而影响查询在被命名查询块中的表上的执行。更多关于查询块名称的信息，请参阅文档 Hint 说明 中的 在 Hint 中定义查询块 小节。

以下是 QB_NAME Hint 的语法：

```
/*+ QB_NAME ( queryblock ) */
```

如果两个或多个查询块具有相同的名称，或者同一个查询块两次被 Hint 指定了不同的名称，那么优化器将忽略所有引用该查询块的名称和 Hint。未使用 QB_NAME Hint 命名的查询块具有由系统生成的惟一名称。这些名称可以显示在计划表中，也可以在查询块中的其他 Hint 中使用。

以下是 QB_NAME Hint 的示例：

```
SELECT /*+ QB_NAME(qb) FULL(@qb e) */ employee_id, last_name
FROM employees e
WHERE last_name = 'Smith';
```

READ_CONSISTENCY Hint

READ_CONSISTENCY Hint 指示服务器去指定某条 SQL 所读取的表模式为弱一致（指定参数 WEAK）或强一致性（指定参数 STRONG）。

以下是 READ_CONSISTENCY Hint 的语法：

```
/*+ READ_CONSISTENCY(WEAK[STRONG]) */
```

示例如下：

```
SELECT /*+ READ_CONSISTENCY(WEAK) */ *
FROM employees
WHERE employees.department_id = 1001;
```

FROZEN_VERSION Hint

FROZEN_VERSION Hint 指示服务器读取某个基线数据的版本。

以下是 FROZEN_VERSION Hint 的语法：

```
/*+ FROZEN_VERSION (intnum) */
```

示例如下：

```
SELECT /*+ FROZEN_VERSION(1000) */ *
FROM employees e
WHERE e.department_id = 1001;
```

QUERY_TIMEOUT Hint

QUERY_TIMEOUT Hint 指示服务器设定某条 SQL 执行时的超时时间，单位为微妙。

以下是 QUERY_TIMEOUT Hint 的语法：

```
/*+ QUERY_TIMEOUT (intnum) */
```

示例如下，当该查询 1 秒之内未执行完该语句即返回超时错误：

```
SELECT /*+ QUERY_TIMEOUT(1000000) */ *
FROM employees e
WHERE e.department_id = 1001;
```

LOG_LEVEL Hint

LOG_LEVEL Hint 指示服务器运行某句 SQL 时采用何种日志级别来执行。

以下是 LOG_LEVEL Hint 的语法：

```
/*+ LOG_LEVEL ([']log_level[']) */
```

以下示例采用 DEBUG 日志级别来执行该 SQL 语句：

```
SELECT /*+ LOG_LEVEL(DEBUG) */ *
FROM employees e
WHERE e.department_id = 1001;
```

USE_PLAN_CACHE Hint

USE_PLAN_CACHE Hint 指示服务器执行某条 SQL 时是否要在计划缓存机制下运行，参数 NONE 为不执行计划缓存机制，参数 DEFAULT 表示按照服务器本身的设置来决定是否执行计划缓存机制。

以下是 USE_PLAN_CACHE Hint 的语法：

```
/*+ USE_PLAN_CACHE (NONE[DEFAULT]) */
```

示例如下，以下语句不执行计划缓存机制：

```
SELECT /*+ USE_PLAN_CACHE(NONE) */ *
FROM employees e
WHERE e.department_id = 1001;
```

TRANS_PARAM Hint

TRANS_PARAM Hint 指示服务器执行事务时是否要按照参数 param 指定的参数来执行，现在支持的参数只有事务层面的提前解行锁 FORCE_EARLY_LOCK_FREE 参数，FORCE_EARLY_LOCK_FREE 的值为 **TRUE** 时表示支持，**FALSE** 表示不支持。注意这里的参数名和参数值要用单引号（ ‘’ ）引起来，当参数的值为数值型时可以不用引号引起来。

以下是 TRANS_PARAM Hint 的语法：

```
/*+ TRANS_PARAM ['param' , 'param_value'] */
```

示例如下：

```
SELECT /*+ TRANS_PARAM('FORCE_EARLY_LOCK_FREE' 'TRUE') */ *
FROM employees e
WHERE e.department_id = 1001;
```

TRACING Hint

TRACING Hint 指示服务器对某些执行计划中的算子采用 TRACING 跟踪。

以下是 TRACING Hint 的语法：

```
/*+ TRACING(TRACING_NUM_LIST)*/
```

示例如下：

```
SELECT /*+ TRACING(1) */ *
FROM employees e
WHERE e.department_id = 1001;
```

STAT Hint

STAT Hint 指示对某些执行计划中的算子采用 STAT 显示信息。

以下是 STAT Hint 的语法：

```
/*+ STAT(TRACING_NUM_LIST) */
```

示例如下：

```
SELECT /*+ STAT(1) */ *
FROM employees e
WHERE e.department_id = 1001;
```

TOPK Hint

TOPK Hint 指示服务器设置模糊查询的精度和最小行数。其中参数 PRECSION 的值为整型，取值范围为 0~100，表示执行模糊查询时的行数百分比，参数 MINIMUM_ROWS 用来指定最小的返回行数。

以下是 TOPK Hint 的语法：

```
/*+ TOPK(PRECISION MINIMUM_ROWS) */
```

示例如下：

```
SELECT /*+ TOPK(1,10) */ *
FROM employees e
WHERE e.department_id = 1001;
```

TRACE_LOG Hint

TRACE_LOG Hint 指示服务器收集跟踪日志（Trace log），收集的跟踪日志（Trace log）在运行 SHOW TRACE 命令时展示。

以下是 TRACE_LOG Hint 的语法：

```
/*+ TRACE_LOG */
```

示例如下：

```
SELECT /*+ TRACE_LOG */ *
FROM employees e
WHERE e.department_id = 1001;
```

4.7 数据库对象

4.7.1 Schema 对象

Schema 是数据的逻辑结构和 Schema 对象的集合。每个 OceanBase 数据库的使用者都拥有一个 Schema，并且 Schema 拥有和使用者相同的名字。

可以使用 SQL 来创建和处理用户数据库对象，并且 Schema 对象有以下类型：

- 约束 Constraints
- 数据库链接 Database links
- 数据库触发器 Database triggers
- 索引 Indexes
- 物化视图 Materialized views
- 对象表 Object tables
- 对象类型 Object types
- 对象视图 Object views
- 包 Packages
- 序列 Sequences
- 存储函数 Stored functions
- 存储过程 Stored procedures
- 同义词 Synonyms
- 表 Tables
- 视图 Views

4.8 数据库对象命名规范

4.8.1 数据库对象命名规范概述

一些数据库对象由用户可以命名的部分和用户必须命名的部分组成。例如，表或视图中的列，索引、表分区和表的子分区，表的完整性约束以及存储在包中的对象（包括程序和存储函数）。本章将讨论：

• 数据库对象命名规则

• Schema 对象命名示例

• Schema 对象命名准则

4.8.2 数据库对象命名规则

数据库对象标识符

每个数据库对象都有名字。在 SQL 语句中您可以使用引用标识符和非引用标识符来命名任何数据库对象：

- 引用标识符：以双引号 "" 为开头和结尾的标识符。如果使用引用标识符命名 Schema 对象，则在引用该对象时必须使用双引号。
- 非引用标识符：标识符中不包含任何标点符号。

但是，数据库名称、全局数据库名称、数据库链接名称、磁盘组名称和可插入数据库（PDB）始终不区分大小写，并以大写形式存储。如果使用引用标识符为它们命名，那么引号将被忽略。

注意 ：OceanBase 并不建议您使用引用标识符来命名数据库对象。这些引用标识符在 SQL PLUS 中兼容，但是在其他数据库对象管理工具中可能不被识别。

标识符使用规则

以下是一些适用于引用标识符和非引用标识符的规则，除非另有说明，否则以下规则对引用标识符和非引用标识符同时适用。

标识符的长度

标识符的长度为 1~128 字节。

如果标识符包含由句点分隔的多个部分，则每个部分最长都可达 128 字节。每个句点分隔符以及旁边的引号都算作一个字节。例如以下字符串:

```
"schema"."table"."column"
```

示例名称中的三部分 **schema**、**table** 和 **column** 各自最长可以是 128 字节，每个引号和句点都是单字节字符，所以示例中标识符的总长度为 392 字节。

保留字做标识符

非引用标识符不能是 OceanBase SQL 保留字。引用标识符可以是保留字，但不建议这样做。根据您计划用于访问数据库对象的工具不同，名称可能进一步受到其他产品特定的保留字限制。

注意：保留字 ROWID 是这个规则的一个例外。不能使用大写单词 ROWID 作为列的名称，无论是带引号的还是不带引号的。但是，引用标识符作为列名时，不可以使用全大写单词，但是可以包含一个或多个小写字母（例如，“Rowid”或“Rowid”）。

特殊含义单词做标识符

OceanBase 中的 SQL 语言包含了其他具有特殊含义的单词。这些词包括数据类型、Schema 名、函数名、虚拟系统表 DUAL 和关键字（SQL 语句中的大写单词，如 DIMENSION、SEGMENT、ALLOCATE、DISABLE 等）。这些词不是保留字，但是，OceanBase 在内部以特定的方式使用它们。因此，如果您使用这些词作为对象和对象部件的名称，SQL 语句可能会难以阅读，并可能导致不可预测的结果。尤其是不要使用 SQL 内置函数的名称作为 Schema 对象和用户自定义函数的名称。

ASCII 字符做标识符

在数据库名称、全局数据库名称和数据库链接名称中您应该使用 ASCII 字符集中的字符，因为这些字符拥有跨不同平台和操作系统的最佳兼容性。多租户容器数据库（CDB）中的通用用户和通用角色的名称只能使用 ASCII 字符集中的字符。

密码中的字符

密码中可以包含多位字节的字符，如汉字或中文标点等。

标识符的开头

非引用标识符必须以数据库字符集中的字母字符开头。引用标识符可以以任何字符开头。

标识符中的符号

非引用标识符只能包含来自数据库字符集的字母数字字符和下划线（_）。但是用于数据库链接名称时可以包含句点（.）和符号 @。引用标识符可以包含任何字符、标点符号和空格。但是，引用标识符和非引用标识符都不能包含双引号或空字符（\0）。

对象名称在名称空间中的限制

在同一个名称空间中，表和视图不能拥有相同的名称。

标识符的大小写

非引用标识符不区分大小写，OceanBase 将他们存储为大写。引用标识符区分大小写，所以通过将名称括在双引号里，在同一命名空间的不同对象可以拥有以下类型的名称：

```
"employees"
"Employees"
"EMPLOYEES"
```

由于非引用标识符不区分大小写，所以 OceanBase 认为以下名称是相同的，他们不能作为同一名称空间下不同对象的名字：

```
employees
EMPLOYEES
"EMPLOYEES"
```

大写标识符

存储或比较大写的标识符时，每个字符的大写形式是通过应用数据库字符集的大写规则来确定的。由对话设置 NLS_SORT 所设定的特定语言规则不在考虑的范围内。此行为是将 SQL 函数 UPPER 应用于标识符，而不是将函

数 NLS_UPPER 应用于标识符。

数据库字符集的大写规则被应用到自然语言时可能产生不正确的结果。例如，德国小写字母 ß，它在数据库字符集大写规则中并没有对应的大写形式， 所以当含有它的标识符被转换为大写形式时它并不会被转变。

数据库字符集大写规则确保标识符在任何语言配置的会话中的样式是相同的。如果希望标识符在某种自然语言中看起来正确，那么可以通过引号来保留它的小写形式，或者在使用该标识符时使用对应语言上正确的大写形式。

列的名称

同一表或视图中的列不能具有相同的名称。但是，不同表或视图中的列可以具有相同的名称。

程序和函数的名称

如果同一包中的程序或函数所包含的参数的数量和数据类型不同，则它们可以具有相同的名称。在同一个包中使用不同的参数创建具有相同名称的多个过程或函数称为重载过程或函数。

4.8.3 Schema 对象命名示例

以下是一些有效的 Schema 对象命名示例：

- last_name
- horse
- hr.hire_date
- "EVEN THIS & THAT!"
- a_very_long_and_valid_name

这些示例都遵循了 数据库对象命名规则 中列出的规则。

4.8.4 Schema 对象命名准则

以下是一些有用的用来命名对象及其部分的准则：

- 使用完整的、描述性的、可发音的名称或众所周知的缩写。
- 使用一致的命名规则。
- 使用相同的名称来描述跨表的相同实体或属性。

在命名对象时，要在简单易用和名称具有描述性的目标之间取得平衡。如果无法平衡，请选择更具描述性的名称，因为数据库中的对象可能会在很长一段时间内被使用，从现在开始十年后的用户可能难以理解名称为 **pmdd** 的表列是用来存储什么信息的，但是 **payment_due_date** 却更清楚明了。

使用一致的命名规则可以帮助用户理解每个表在应用程序中扮演的角色。例如，以 **fin_** 为开头命名所有属于 **Finance** 应用程序的表的名称。

使用相同的名称描述表中的相同内容。例如，雇员和部门表的部门号列其实存储的是一样的值，那么在雇员表和部门表中这一列均被命名为 **department_id**。

4.9 数据库对象引用方式

4.9.1 数据库对象引用概述

本章将讨论如何在 SQL 语句的上下文中引用 Schema 对象及其部分。本章包括：

- 引用 Schema 对象
- 引用远程数据库中的对象
- 引用分区表和索引
- 引用对象类型属性和方法

更多信息

- 引用远程数据库链接
- 引用其他 Schema 中的对象

4.9.2 引用 Schema 对象

当在 SQL 语句中引用对象时，OceanBase 会考虑 SQL 语句的上下文并在适当的名称空间中找到该对象。找到对象后，OceanBase 执行 SQL 语句对该对象指定的操作。如果在适当的名称空间中找不到命名的对象，则返回错误。

引用用户的 Schema 中的对象

以下示例说明了 OceanBase 如何解析 SQL 语句中对象的引用。执行以下语句，向名为 **departments** 的表中添加了一行数据：

```
INSERT INTO departments VALUES (280, 'ENTERTAINMENT_CLERK', 206, 1700);
```

根据 SQL 语句的上下文，**departments** 在数据库中可能是：

- 一张在您自己 Schema 中的表
- 一个在您自己 Schema 中的视图
- 一个表或视图的专有同义词
- 一个公共同义词

OceanBase 始终会先尝试在您自己的 Schema 中的名称空间里解析被引用对象，然后再考虑该 Schema 之外的名称空间。在此示例中，OceanBase 尝试解析如下：

首先，OceanBase 尝试在您自己的包含了表、视图和专用同义词的 Schema 中的名称空间里去定位对象。如果对象是专用同义词，则 OceanBase 会找到该同义词所代表的对象。该对象可以在您自己的 Schema 中，也可以在其他 Schema 中或者在另一个数据库中。该对象也可以是其他的同义词，在这种情况下，OceanBase 会找到同义词所代表的对象。

如果对象在名称空间中被找到，则 OceanBase 尝试对该对象执行 SQL 语句。在此示例中

，OceanBase 尝试将一行数据添加到 **departments** 中。如果此对象的类型不是该 SQL 语句所需要的类型，则 OceanBase 返回错误。在此示例中，**departments** 必须是表、视图或者可以是可以解析为表或视图的专用同义词。如果 **departments** 是序列，则 OceanBase 返回错误。

如果该对象到目前为止没有在任何名称空间中被搜索到，则 OceanBase 将搜索包含公共同义词的名称空间。如果对象在包含公共同义词的名称空间中，则 OceanBase 尝试对该对象执行该语句。如果此对象的类型不是该 SQL 语句所需要的类型，则 OceanBase 返回错误，例如在此示例中，如果 **departments** 是一个代表了序列的公共同义词，那么 OceanBase 将返回错误。

如果公共同义词包含任何依赖表或用户定义类型，则不能在与依赖对象相同的 Schema 中创建与同义词同名的对象。

反之，如果同义词没有任何依赖表或用户定义的类型，则可以在与依赖对象相同的 Schema 中创建具有相同名称的对象。OceanBase 会使所有依赖对象无效，并在下一次访问它们时重新验证它们。

引用其他 Schema 中的对象

要引用您的 Schema 之外的 Schema 中的对象，需要在对象名称前添加 Schema 名称：

```
schema.object
```

例如，以下示例展示了在名为 **hr** 的 Schema 中删除表 **employees**：

```
DROP TABLE hr.employees;
```

4.9.3 引用远程数据库中的对象

要引用除本地数据库以外的其他数据库中的对象，需要在对象名称后加上指向该数据库的数据库链接的名称。数据库链接是一个 Schema 对象，它使 OceanBase 连接到远程数据库以访问里面的对象。

创建数据库链接

可以使用语句 CREATE DATABASE LINK 创建数据库链接，使用该语句时需要指定以下数据库链接的信息，OceanBase 将下述信息存储在数据字典中：

- 租户名、用户名和密码。
- 网络地址、端口号。

创建链接限制

1. 只支持创建 OceanBase 中的 Oracle 模式租户连接到另一个 Oracle 模式租户的数据库链接，OceanBase 目前不支持创建连接到外部 Oracle 数据库的数据库链接。
2. 一个租户可以反复创建、删除数据库链接，但最多只能同时存在 15 个数据库链接。

数据库连接串格式

```
user_name@tenant_name IDENTIFIED BY password HOST 'ip:port'
```

- **user_name**：用户名。
- **tenant_name**：租户名。
- **password**：密码。
- **ip**：网络地址。
- **port**：端口号。

以下是创建数据库链接的示例：

```
CREATE DATABASE LINK my_link CONNECT TO
root@oracle IDENTIFIED BY abcdef HOST '192.168.0.0:1521';
```

引用数据库链接

当您发出包含数据库链接的 SQL 语句时，需要使用存储在数据字典中的完整数据库链接名。

OceanBase 在连接远程数据库时，将使用数据库连接字符串访问远程数据库。使用数据库字符串、用户名和密码后连接成功，则 OceanBase 使用文档 引用 Schema 对象 中的规则来访问远程数据库上的指定对象，但请注意相关限制：

1. 只能执行只读语句。
2. 只能访问表对象，不支持访问其它对象，如视图、序列等。
3. 访问表对象时必须显式指定数据库名，如：test.t1@my_link。
4. 不支持部分计划和算子：
    - 不能在远程数据库执行 RESCAN 操作。
    - 不能在远程数据库执行 NESTED LOOP JOIN、SEMI JOIN、ANTI JOIN、SUBPLAN FILTER 等算子。
    - 如遇以上问题请尝试运行 EXPLAIN PLAN 语句查看原始计划和发送到远端集群执行的 SQL 语句，并通过使用 Hint 调整计划。

4.9.4 引用分区表和索引

表和索引可以分区。进行分区时，这些 Schema 对象由许多称为分区的部分组成，所有这些部分都具有相同的逻辑属性。例如，表中的所有分区共享相同的列和约束定义，而索引中的所有分区共享相同的索引列。

分区扩展名和子分区扩展名使用户可以执行某些分区级和子分区级操作，例如，仅在一个分区或子分区上删除其中的所有行。没有扩展名称时，此类操作将要求您使用判断语句定义范围（WHERE 子句）。对于范围分区表和列表分区表，尝试用判断语句描述分区级操作可能会很麻烦，尤其是当范围分区键使用多个列时。对于哈希分区和子分区，使用判断语句更加困难，因为这些分区和子分区是基于系统定义的哈希函数。

分区扩展名使您可以像使用表一样使用分区。此方法的一个优点（对范围分区的表最有用）是，您可以通过对其他用户或角色授予（或撤消）这些视图的特权来构建分区级别的访问控制机制。要将分区用作表，需要通过从单个分区中选择数据来创建视图，然后将该视图用作表。

语法

当在 SQL 语句的语法或轨道图中出现 partition_extended_name 或者 subpartition_extended_name 元素时，用户可以通过这两个元素指定分区扩展表名和子分区扩展表名。

partition_extended_name 的语法：

```
PARTITION partition
|
PARTITION FOR ( partition_key_value [, partition_key_value]... )
```

subpartition_extended_name 的语法：

```
SUBPARTITION subpartition
|
SUBPARTITION FOR ( subpartition_key_value [, subpartition_key_value]... )
```

DML 语句 INSERT、UPDATE 、 DELETE 和 ANALYZE 中要求在分区或子分区名称的周围加上括号。这个小的区别体现在 partition_extension_clause 元素中。

partition_extension_clause 的语法：

```
{ PARTITION (partition)
| PARTITION FOR (partition_key_value [, partition_key_value]...)
| SUBPARTITION (subpartition)
| SUBPARTITION FOR (subpartition_key_value [, subpartition_key_value]...)
}
```

在 partition_extended_name，subpartition_extended_name 和 partition_ extension_clause 中，可以使用 PARTITION FOR 和 SUBPARTITION FOR 子句在不使用名称的情况下引用分区。它们对任何类型的分区均有效，尤其是间隔分区。将数据插入表中时，间隔分区会根据需要被自动创建。

对于上述元素中各自的 partition_key_value 或 subpartition_key_value 部分，它们为每个分区键列指定一个值。对于多列分区键，需要为每个分区键指定一个值。对于复合分区，对每个分区键指定一个值后，需要继续为每个子分区键指定一个值。所有分区键值均以逗号分隔。对于间隔分区，您只能指定一个 partition_key_value，并且它必须是有效的 NUMBER 数据类型或日期时间数据类型的值。用户的 SQL 语句将在包含用户指定了值的分区或子分区上运行。

扩展名称的限制

使用分区扩展表名和子分区扩展表名时有以下限制：

- 名称中没有远程表：扩展分区表名和子分区扩展表名不能包含数据库链接或能转换为具有数据库链接的表的同义词。要使用远程分区和远程子分区，需要在远程站点上使用扩展表名语法创建一个视图，然后引用该远程视图。
- 名称中没有同义词：必须使用基表指定分区或子分区扩展名，不能使用同义词，视图或任何其他对象。
- 在 PARTITION FOR 和 SUBPARTITION FOR 子句中，不能指定关键字 DEFAULT 、MAXVALUE 或绑定变量为 partition_key_value 或 subpartition_key_value 的值。

- 在 PARTITION 和 SUBPARTITION 子句中，不能为分区或子分区名称指定绑定变量。

示例

在以下示例中，**sales** 是具有分区 **sales_q1_2000** 的分区表。以下语句创建了分区 **sales_q1_2000** 的视图，然后像使用表一样使用它。本示例从分区中删除了一些行：

```
/*为分区 sales_q1_2000 创建视图 Q1_2000_sales*/
CREATE VIEW Q1_2000_sales
AS
SELECT *
FROM sales PARTITION (SALES_Q1_2000);
/*删除视图 Q1_2000_sales 中符合条件 amount_sold < 0 的值*/
DELETE FROM Q1_2000_sales
WHERE amount_sold < 0;
```

4.9.5 引用对象类型属性和方法

要在 SQL 语句中引用对象类型属性或方法，必须使用表别名完全限定该引用。以下示例中样本 Schema **ob** 包含类 **cust_address_typ** 和表 **customers**，**customers** 拥有一个 **cust_address_typ** 类型的列 **cust_address**：

```
CREATE TYPE cust_address_typ
OID '82A4AF6A4CD1656DE034080020E0EE3D'
AS OBJECT
(street_address VARCHAR2(40),
postal_code VARCHAR2(10),
city VARCHAR2(30),
state_province VARCHAR2(10),
country_id CHAR(2));
/
CREATE TABLE customers
(customer_id NUMBER(6),
cust_first_name VARCHAR2(20) CONSTRAINT cust_fname_nn NOT NULL,
cust_last_name VARCHAR2(20) CONSTRAINT cust_lname_nn NOT NULL,
cust_address cust_address_typ,
. . .
```

在 SQL 语句中，对 **postal_code** 属性的引用必须使用表别名进行完全限定，如下所示：

```
SELECT c.cust_address.postal_code
FROM customers c;

UPDATE customers c
SET c.cust_address.postal_code = '610000'
WHERE c.cust_address.city = 'chengdu'
AND c.cust_address.state_province = 'SICHUAN';
```

要引用不接受参数的成员方法，必须提供空括号。例如，样本 Schema **ob** 包含一个基于 **catalog_typ** 的对象表 **category_tab**，该表包含成员函数 getCatalogName。为了在 SQL 语句中调用此方法，必须提供空括号，如下所示：

```
SELECT TREAT(VALUE(c) AS catalog_typ).getCatalogName()"Catalog Type"
FROM categories_tab c
WHERE category_id = 10;
```

返回结果：

```
+----------------+
| Catalog Type |
+----------------+
| online catalog |
+----------------+
```

5 运算符

5.1 运算符概述

运算符一般用于连接运算数或参数之类的单个数据项且返回结果。从语法上讲，运算符出现在运算数之前、之后或两个运算数之间均可。一般用特殊字符或关键字表示运算符，例如，除法运算符用斜杠（/）表示。本章讨论非逻辑（非布尔）运算符，这些运算符本身不能用作查询或子查询的 WHERE 或 HAVING 条件。

本章将详细介绍以下运算符：

- 算术运算符
- 串联运算符
- 层次查询运算符
- 集合运算符

一元和二元运算符

运算符可分为两大类:

- 一元运算符：一元运算符仅对一个运算数进行运算。一元运算符常用格式：

  ```
  运算符 运算数
  ```

- 二元运算符：二元运算符顾名思义是对两个运算数进行运算。二元运算符常用格式：

  ```
  运算数1 运算符 运算数2
  ```

  其他具有特殊格式的运算符可接受两个以上的运算数。如果为运算符提供了空运算数， 则结果始终为空。唯一不遵循此规则的是串联（||）运算符。

运算符优先级

优先级会影响 OceanBase 数据库在同一表达式中评估不同运算符的顺序。在评估包含多个运算符的表达式时，OceanBase 会先评估优先级更高的运算符，然后再评估优先级较低的运算符，优先级相等时，在表达式中从左到右对相等的运算符求值。下表列出了 SQL 运算符中从高到低的优先级。同一行上列出的运算符具有相同的优先级。

运算符	运算方式
+、-（作为一元运算符）、PRIOR、CONNECT_BY_ROOT	身份、否定、层次结构中的位置。
*、/	乘法、除法。
+、-（作为二进制运算符）、||	加、减、串联。
条件表达式在数据库评估完运算符后再进行评估	详情请参阅文档 条件优先 。

在以下示例中，由于乘法的优先级高于加法，因此数据库首先将 2 乘以 3，然后再将结果加到 1。

```
1+2*3
```

您也可以在表达式中使用括号来限制运算符优先级。OceanBase 将先对括号内的表达式求值，再对括号外部的表达式求值。

SQL 还支持集合运算符（UNION、UNION ALL、INTERSECT 和 MINUS），集合运算符会合并查询返回的行集，而不是单个数据项。所有集合运算符都具有相同的优先级。

5.2 运算符列表

5.2.1 算术运算符

算术运算符用来对一个或两个参数进行求反、加、减、乘和除等操作。其中一些算术运算符还可用于计算日期时间和间隔值。算术运算符的参数必须解析为数字数据类型或任何可以被数据库直接转换为数字数据类型的数据类型。

一元算术运算符返回的数据类型与参数的数据类型相同。对于二进制算术运算符， OceanBase 先确定表达式中数值优先级最高的参数，再将其余参数转换为该参数的数据类型。

下表列出了算术运算符：

运算符	说明
+、-	用于表示正、负时，它们是一元运算符。
+、-	用于表示加、减时，它们是二进制运算符。
*、/	二进制运算符，用于表示乘、除。

算术表达式中不能使用两个连续的负号（- -）来表示双重否定或减去一个负值，因为字符两个连续的负号在 SQL 语句中用于指定注释。可以用空格或括号分隔连续的减号。有关 SQL 语句中的注释，详细信息请参阅章节 注释 。

以下示例语句展示了运算符 +、- 用于表示正、负的 SQL 查询：

```
SELECT * FROM order_items WHERE quantity = -1 ORDER BY order_id, line_item_id, product_id;
SELECT * FROM employees WHERE -salary < 0 ORDER BY employee_id;
```

以下示例语句展示了运算符 +、- 用于表示加、减的 SQL 查询：

```
SELECT hire_date FROM employees WHERE SYSDATE - hire_date > 365 ORDER BY hire_date;
```

以下示例语句展示了 *、/ 运算符表示乘、除的 SQL 查询：

```
UPDATE employees SET salary = salary * 1.1;
UPDATE employees SET salary = salary / 2;
```

5.2.2 串联运算符

串联运算符 || 用于连接字符串和 CLOB 数据类型的数据。

连接两个字符串后会得到另一个字符串。如果两个字符串的数据类型均为 CHAR，则返回结果的数据类型也为 CHAR，并且限制为 2000 个字符。如果被合并的任一字符串的数据类型为 VARCHAR2，则返回结果的数据类型也为 VARCHAR2，并且限制为 4000 个字符。如果串联运算符两边的参数中任何一个是 CLOB 数据类型的值，则返回结果的数据类型为临时 CLOB。无论两边参数的数据类型是什么，字符串中的尾随空格都是通过串联保存的。

尽管 OceanBase 将长度为 0 的字符串视为空值，但是将一个带有值的操作数和另一个 0 长度字符串连接得到的结果是一个带值的操作数，因此 NULL 只能由两个值为 NULL 的字符串串联产生。但是，在将来的 OceanBase 数据库版本中，情况可能不会继续如此。要连接可能为空的表达式，请使用 函数将表达式直接转换为长度为 0 的字符串。

以下示例创建了一个同时包含 CHAR 和 VARCHAR2 类型列的表，然后插入带有和不带有尾随空格的值，并将它们连接起来。

```
CREATE TABLE tab1 (col1 VARCHAR2(6), col2 CHAR(10), col3 VARCHAR2(10), col4 CHAR(6));
INSERT INTO tab1 (col1, col2, col3, col4) VALUES ('abc', 'def ', 'ghi ', 'jkl');
SELECT col1 || col2 || col3 || col4"Concatenation"FROM tab1;
```

返回结果：

```
+------------------+
|Concatenation |
+------------------+
|abcdef ghi jkl|
+------------------+
```

5.2.3 层次查询运算符

PRIOR 和 CONNECT_BY_ROOT 运算符仅在 层次查询 中有效。

PRIOR 运算符

在层次查询中，CONNECT BY 条件中的一个表达式必须由 PRIOR 运算符限定。如果 CONNECT BY 条件是复合条件，只有一个条件要求使用 PRIOR 运算符，但是也可以有多个包含 PRIOR 运算符的条件。PRIOR 在层次查询中为当前行的父行计算紧随其后的表达式。

PRIOR 最常用在比较列值与相等运算符时（PRIOR 关键字可以在运算符的任意一 侧）。PRIOR 使数据库使用列中父行的值。理论上，在 CONNECT BY 子句中可以使用除等号（=）以外的其他运算符。但是，由这些其他运算符创建的条件可能会导致无限循环，在这种情况下，OceanBase 在运行时去检测循环并返回错误。

CONNECT_BY_ROOT 运算符

CONNECT_BY_ROOT 是一元运算符，仅在层次查询中有效。使用此运算符限定列时，OceanBase 使用根行中的数据返回列值。该运算符扩展了层次查询的 CONNECT BY [PRIOR] 条件的功能。

您不能在 START WITH 条件或 CONNECT BY 条件中指定此运算符。

5.2.4 集合运算符

集合运算符将两个查询结果合并为一个结果。包含集合运算符的查询称为复合查询。

运算符	返回值
UNION	返回任意查询选择的所有不同行。
UNION ALL	返回任意查询选择的所有行，并包括所有重复项。
INTERSECT	返回两个查询选择的所有不同行。
MINUS	返回第一个查询选择的所有不同行，但不返回第二个查询选择的所有行。

关于集合运算符的更多信息和示例，请参阅查询和子查询章节 集合 文档中关于 UNION ALL，INTERSECT，MINUS 运算符的介绍。

6 函数

6.1 函数概述

函数与运算符相似，即给定一些数据元素作为参数的输入并返回结果。然而函数在入参形式上与运算符有比较大的区别。函数允许包含的参数数量不定，一个函数里可以有一到两个，甚至更多的参数。

```
函数名（参数，参数，...）
```

没有任何参数的函数类似于 伪列 。但是，伪列通常为结果集中的每一行返回不同的值，而没有任何变量的函数通常为每一行返回相同的值。

关于函数

OceanBase 内嵌的函数可以直接在 SQL 语句中使用。每个函数的入参传入值均有期望的数据类型，如果传入的数据类型不是期望的数据类型，则 OceanBase 会在实际执行 SQL 函数之前尝试将参数传入的数值转换为期望的数据类型。

函数中的空值

对于绝大多数的函数当入参为空值 NULL 的时候，其返回的结果也为 NULL。这种情况下，您可以使用 NVL 函数返回一个非空值。例如，一张记录佣金的表的佣金列 **commission_pct** 为空值 NULL ，则表达式 NVL(commission_pct，0) 返回 0；如果 **commission_pct** 的值不为 NULL，则返回实际的佣金值。

函数分类与列表

在如下的函数分类列表中，每一类函数的参数和最终的函数返回值都有其特定的数据类型。

注意： 在 SQL 语句中对 LOB 列使用函数时，OceanBase 数据库将在 SQL 和 PL/SQL 处理期间创建临时 LOB 列，并有一定的使用限制，详情信息请参考文档 与 Oracle 兼容性对比 。

本章中函数分成了两大类：

- **单行函数**：包括 数字函数 、 返回字符串的字符串函数 、 返回数字的字符串函数 、 时间日期函数 、 通用比较函数 、 转换函数 、 编码解码函数 和 空值相关函数 。
- **统计函数**：包括 聚合函数 和 分析函数 。

单行函数对于被查询的表或者视图每一行均返回一个结果值，这些函数可以使用在 SQL 语句的 SELECT、WHERE 、START WITH、CONNECT BY、HAVING 等子句当中。

分析函数与聚合函数，都是对行集组（一组行的集合）进行聚合计算，不同的是，聚合函数每组只能返回一个值（一行），而分析函数每组可以返回多个值（多行）。行集组又称为窗口（Window）。聚合函数通常和 SELECT 语句中的 GROUP BY 子句一起使用，使用时数据库将查询表或视图的行分为几组，并将聚合函数应用于每组行，同时为每组返回一个结果行。

使用分析函数时需要用特殊的关键字 OVER 来指定窗口。更多关于窗后函数的信息，请参阅文档 窗口函数说明 。

数字函数

数字函数的变量输入与函数输出结果均为数字类型，绝大部分的数字函数的返回值的数据类型为 NUMBER，可以精确到小数点后 38 位。

一些高等代数相关函数 COS，COSH，EXP，LN，LOG，SIN，SINH，SQRT，TAN，TANH 等函数的结果精确到小数点后 36 位，其他代数相关函数 ACOS，ASIN，ATAN 和 ATAN2其结果返回值精确到小数点后 30 位。

函数分类	函数子分类	函数名	功能描述
单行函数	数字函数	ABS	返回指定数值表达式的绝对值（正值）的数学函数。
单行函数	数字函数	ACOS	返回以弧度表示的角，其余弦为指定的 NUMBER 表达式，也称为反余弦。
单行函数	数字函数	ASIN	OceanBase 暂不支持。
单行函数	数字函数	ATAN	OceanBase 暂不支持。

单行函数	数字函数	ATAN2	OceanBase 暂不支持。
单行函数	数字函数	BITAND	运算符按位进行“与”操作。输入和输出类型均为 NUMBER 数据类型。
单行函数	数字函数	CEIL	返回值大于等于数值 numeric_expression 的最小整数。
单行函数	数字函数	COS	OceanBase 暂不支持。
单行函数	数字函数	COSH	OceanBase 暂不支持。
单行函数	数字函数	EXP	返回 e 的 numeric_expression 次幂。
单行函数	数字函数	FLOOR	返回小于等于数值 numeric_expression 的最大整数。
单行函数	数字函数	LN	返回以 e 为底的 numeric_expression 的对数。
单行函数	数字函数	LOG	返回以 x 为底的 y 的对数。
单行函数	数字函数	MOD	返回 x 除以 y 的余数。
单行函数	数字函数	POWER	返回 x 的 y 次幂。
单行函数	数字函数	REMAINDER	返回 x 除以 y 的余数。
单行函数	数字函数	ROUND	返回 numeric 四舍五入后的值。
单行函数	数字函数	SIGN	返回数字 n 的符号，大于 0 返回 1，小于 0 返回 -1 ，等于 0 返回 0。
单行函数	数字函数	SIN	OceanBase 暂不支持。
单行函数	数字函数	SINH	OceanBase 暂不支持。
单行函数	数字函数	SQRT	返回 n 的平方根。
单行函数	数字函数	TAN	OceanBase 暂不支持。
单行函数	数字函数	TANH	OceanBase 暂不支持。
单行函数	数字函数	TRUNC	返回 numeric 按精度 precision 截取后的值。
单行函数	数字函数	WIDTH_BUCKET	OceanBase 暂不支持。

返回字符串的字符串函数

函数的返回值的最大长度受数据类型的影响，比如：函数的返回值的数据类型是 VARCHAR2，但是返回值实际的大小超过了 VARCHAR2 数据类型的最大限制，此时 OceanBase 数据库会对结果进行截断处理并返回，但是在客户端上并不会显示提示。

注意： 如果返回值的数据类型是 CLOB，当返回值长度超过了最大限制时，OceanBase 不会返回数据且显示错误提示。

函数分类	函数子分类	函数名	功能描述
单行函数	返回字符串的字符串函数	CHR	将 n 转换为等价的一个或多个字符返回，且返回值与当前系统的字符集相关。
单行函数	返回字符串的字符串函数	CONCAT	连接两个字符串。
单行函数	返回字符串的字符串函数	INITCAP	返回字符串并将字符串中每个单词的首字母大写，其他字母小写。
单行函数	返回字符串的字符串函数	LOWER	将字符串全部转为小写。
单行	返回字符串的字符	LPAD	在字符串 c1 的左边用字符串 c2 填充，直到长度为 n 时为止。

函数	串函数		
单行函数	返回字符串的字符串函数	LTRIM	删除左边出现的字符串。
单行函数	返回字符串的字符串函数	REGEXP_REPLACE	用于正则表达式替换。
单行函数	返回字符串的字符串函数	REGEXP_SUBSTR	OceanBase 暂不支持。
单行函数	返回字符串的字符串函数	REPLACE	将字符表达式值中，部分相同字符串，替换成新的字符串。
单行函数	返回字符串的字符串函数	RPAD	在字符串 c1 的右边用字符串 c2 填充，直到长度为 n 时为止。
单行函数	返回字符串的字符串函数	RTRIM	删除右边出现的字符串，此函数对于格式化查询的输出非常有用。
单行函数	返回字符串的字符串函数	SUBSTR	截取子字符串。其中多字节符（汉字、全角符等）按 1 个字符计算。
单行函数	返回字符串的字符串函数	TRANSLATE	将字符表达式值中，指定字符替换为新字符。多字节符（汉字、全角符等），按 1 个字符计算。
单行函数	返回字符串的字符串函数	TRIM	删除一个字符串的开头或结尾（或两者）的字符。
单行函数	返回字符串的字符串函数	UPPER	将字符串全部转为大写。

返回数字的字符串函数

函数分类	函数子分类	函数名	功能描述
单行函数	返回数字的字符串函数	ASCII	返回字符表达式最左端字符的 ASCII 码值。
单行函数	返回数字的字符串函数	INSTR	在一个字符串中搜索指定的字符，返回发现指定的字符的位置。
单行函数	返回数字的字符串函数	LENGTH	返回字符串的长度。
单行函数	返回数字的字符串函数	REGEXP_COUNT	OceanBase 暂不支持。
单行函数	返回数字的字符串函数	REGEXP_INSTR	OceanBase 暂不支持。

时间日期函数

时间日期函数支持的入参数据类型有三类：日期时间 (DATE)，时间戳 (TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE)，以及间隔 (INTERVAL DAY TO SECOND、INTERVAL YEAR TO MONTH)。

仅支持 DATE 数据类型入参的函数列表：ADD_MONTHS、CURRENT_DATE, LAST_DAY、NEW_TIME 和 NEXT_DAY。如果对于以上函数，您尝试给入的是 TIMESTAMP 类型的数据，OceanBase 内部会先进行隐式数据类型转换后带入函数进行运算，并返回 DATE 类型的返回值。

注意：MONTHS_BETWEEN 返回值为 1 个数字。ROUND 和 TRUNC 并不能进行隐式转换，必须传输 DATE 类型的数值，否则会报错。

剩余的函数对于三种参数数据类型均可以支持，且返回和入参同样的数据类型。

函数分类	函数子分类	函数名	功能描述
单行函数	时间日期函数	ADD_MONTHS	返回在日期 date 基础上 n 个月后的日期值，如果 n 的值为负数则返回日期 date 基础上 n 个月前的日期值（date 减去 n 个月）。
单行函数	时间日期函数	CURRENT_DATE	返回当前会话时区中的当前日期。
单行函数	时间日期函数	CURRENT_TIMESTAMP	返回 TIMESTAMP WITH TIME ZONE 数据类型的当前会话时区中的当前日期，返回值中包含当前的时区信息。
单行函数	时间日期函数	DBTIMEZONE	返回当前数据库实例的时区，在 OceanBase 中数据库时区恒为 GMT+00:00，且不支持修改。
单行函数	时间日期函数	EXTRACT（datetime)	从指定的时间字段或表达式中抽取年、月、日、时、分、秒等元素。
单行函数	时间日期函数	FROM_TZ	将一个 TIMSTAMP 数据类型的值和时区信息拼成一个 TIMESTAMP WITH TIME ZONE 数据类型的时间值。
单行函数	时间日期函数	LAST_DAY	返回日期 date 所在月份的最后一天的日期。
单行函数	时间日期函数	LOCALTIMESTAMP	返回当前会话时区中的当前日期，返回 TIMESTAMP 数据类型的值。
单行函数	时间日期函数	MONTHS_BETWEEN	返回返回参数 date1 到 date2 之间的月数。
单行函数	时间日期函数	NEW_TIME	OceanBase 暂不支持。
单行函数	时间日期函数	NEXT_DAY	返回日期 d1 的下一周中 c1（星期值）所在的日期值。
单行函数	时间日期函数	NUMTODSINTERVAL	把参数 n 转为以参数 interval_unit 为单位的 INTERVAL DAY TO SECOND 数据类型的值。
单行函数	时间日期函数	NUMTOYMINTERVAL	把参数 n 转为以 interval_unit 为单位的 INTERVAL YEAR TO MONTH 数据类型的值。
单行函数	时间日期函数	ROUND (date)	返回以参数 fmt 为单位距离的离指定日期 date 最近的日期时间值。
单行函数	时间日期函数	SESSIONTIMEZONE	返回当前会话时区。
单行函数	时间日期函数	SYS_EXTRACT_UTC	返回与指定时间相对应的标准 UTC 时间。
单行函数	时间日期函数	SYSDATE	返回当前日期。
单行函数	时间日期函数	SYSTIMESTAMP	返回系统当前日期，返回值的秒的小数位包含 6 位精度，且包含当前时区信息。
单行函数	时间日期函数	TO_CHAR(datetime)	将 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 等数据类型的值按照参数 fmt 指定的格式转换为 VARCHAR2 数据类型的值。
单行函数	时间日期函数	TO_DSINTERVAL	将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL DAY TO SECOND 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。
单行函数	时间日期函数	TO_TIMESTAMP	将字符串转换为 TIMESTAMP 数据类型。
单行函数	时间日期函数	TO_TIMESTAMP_TZ	将字符串转换为 TIMESTAMP WITH TIME ZONE 数据类型，包含时区信息。

单行函数	时间日期函数	TO_YMINTERVAL	将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL YEAR TO MONTH 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。
单行函数	时间日期函数	TRUNC(date)	返回以参数 fmt 为单位距离的离指定日期 date 最近的日期时间值，并且返回的日期值在 date 之前。
单行函数	时间日期函数	TZ_OFFSET	返回时区 n 的时区偏移量。时区偏移量是指与格林尼治标准时间 GMT 的差（小时和分钟）。

通用比较函数

可以通过本类别函数快速的在集合中寻找到最大值和最小值。

函数分类	函数子分类	函数名	功能描述
单行函数	通用比较函数	GREATEST	返回一个或多个表达式列表中的最大值。
单行函数	通用比较函数	LEAST	返回一个或多个表达式列表中的最小值。

转换函数

可以通过本类型的函数将原本的数据类型转换为另外一种数据类型。

函数分类	函数子分类	函数名	功能描述
单行函数	转换函数	ASCIISTR	OceanBase 暂不支持。
单行函数	转换函数	BIN_TO_NUM	OceanBase 暂不支持。
单行函数	转换函数	CAST	用于将源数据类型的表达式显式转换为另一种数据类型。
单行函数	转换函数	CHARTOROWID	OceanBase 暂不支持。
单行函数	转换函数	HEXTORAW	将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型中包含十六进制数字的字符转换为 RAW 数据类型。
单行函数	转换函数	RAWTOHEX	将二进制数转换为一个相应的十六进制表示的字符串。
单行函数	转换函数	TO_BINARY_DOUBLE	返回一个双精度的 64 位浮点数.
单行函数	转换函数	TO_BINARY_FLOAT	返回一个单精度的 32 位浮点数。
单行函数	转换函数	TO_CHAR(character)	将 NCHAR、NVARCHAR2 或 CLOB 数据转换为数据库字符集。
单行函数	转换函数	TO_CHAR(datetime)	将 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 等数据类型的值按照参数 fmt 指定的格式转换为 VARCHAR2 数据类型的值。
单行函数	转换函数	TO_CHAR(number)	将 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 类型的数值 n 按照指定数值格式 fmt 转换为 varchar2 数据类型的值。
单行函数	转换函数	TO_DATE	将 CHAR、VARCHAR、NCHAR 或 NVARCHAR2 数据类型的字符转换为日期数据类型的值。
单行函数	转换函数	TO_DSINTERVAL	将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL DAY TO SECOND 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。

单行函数	转换函数	TO_NUMBER	将 expr 转换为数值数据类型的值。
单行函数	转换函数	TO_TIMESTAMP	将字符串转换为 TIMESTAMP 数据类型。
单行函数	转换函数	TO_TIMESTAMP_TZ	将字符串转换为 TIMESTAMP WITH TIME ZONE 数据类型，包含时区信息。
单行函数	转换函数	TO_YMINTERVAL	将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL YEAR TO MONTH 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。

编码解码函数

可以通过本类型的函数在 OceanBase 数据库中实现数据的编码解密需求。

函数分类	函数子分类	函数名	功能描述
单行函数	编码解码函数	DECODE	会根据条件返回相应值。
单行函数	编码解码函数	ORA_HASH	获取对应表达式的 HASH 值。
单行函数	编码解码函数	VSIZE	返回 X 的字节大小数。

空值相关函数

可以通过本类型的函数在 OceanBase 数据库中对 NULL 值进行相关的处理。

函数分类	函数子分类	函数名	功能描述
单行函数	空值相关函数	COALESCE	返回参数列表中第一个非空表达式，必须指定最少两个参数。
单行函数	空值相关函数	LNNVL	判断条件中的一个或者两个操作数是否为 NULL。也可以将条件作为参数，如果条件为 FALSE 或 UNKNOWN，则返回 TRUE；如果条件为 TRUE，则返回 FALSE。
单行函数	空值相关函数	NULLIF	OceanBase 暂不支持。
单行函数	空值相关函数	NVL	从两个表达式返回一个非 NULL 值。如果 expr1 与 expr2 的结果皆为 NULL 值，则 NVL 函数返回 NULL。
单行函数	空值相关函数	NVL2	根据表达式是否为空，返回不同的值。如果 expr1 不为空，则返回 expr2 的值，如果 expr1 为空，则返回 expr3 的值。expr2 和 expr3 类型不同的话，expr3 会转换为 expr1 的类型。

环境相关函数

本分类的函数主要提供 Session 或者租户实例相关的环境信息。

函数分类	函数子分类	函数名	功能描述
单行函数	环境相关函数	SYS_CONTEXT	OceanBase 暂不支持。
单行函数	环境相关函数	UID	OceanBase 暂不支持。
单行函数	环境相关函数	USER	OceanBase 暂不支持。

聚合函数

函数分类	函数子分类	函数名	功能描述
统计函数	聚合函数	AVG	返回数值列的平均值。
统计函数	聚合函数	COUNT	用于查询参数 expr 的行数。
统计函数	聚合函数	SUM	返回参数中指定列的和。
统计函数	聚合函数	GROUPING	OceanBase 暂不支持。
统计函数	聚合函数	MAX	返回参数中指定的列中的最大值。
统计函数	聚合函数	MIN	返回参数中指定列的最小值。
统计函数	聚合函数	LISTAGG	用于列转行，LISTAGG 对 ORDER BY 子句中指定的每个组内的数据进行排序，然后合并度量列的值。
统计函数	聚合函数	ROLLUP	在数据统计和报表生成过程中，它可以为每个分组返回一个小计，同时为所有分组返回总计。
统计函数	聚合函数	STDDEV	用于计算总体标准差。
统计函数	聚合函数	STDDEV_POP	计算总体标准差。
统计函数	聚合函数	STDDEV_SAMP	计算样本标准差。
统计函数	聚合函数	VARIANCE	返回参数指定列的方差。
统计函数	聚合函数	APPROX_COUNT_DISTINCT	计算某一列去重后的行数，返回的值是一个近似值，该函数可以进一步用于计算被引用的列的选择性。

分析函数

函数分类	函数子分类	函数名	功能描述
统计函数	分析函数	AVG	返回数值列的平均值。
统计函数	分析函数	COUNT	用于查询参数 expr 的行数。
统计函数	分析函数	CUME_DIST	计算一个值在一组值中的累积分布。
统计函数	分析函数	DENSE_RANK	计算有序行组中行的秩，并将秩作为 NUMBER 返回。
统计函数	分析函数	MAX	返回参数中指定的列中的最大值。
统计函数	分析函数	MIN	返回参数中指定列的最小值。
统计函数	分析函数	SUM	返回参数中指定列的和。
统计函数	分析函数	FIRST_VALUE	返回有序值中的第一个值。

统计函数	分析函数	LAG	提供对多行表的访问，而不需要自连接。
统计函数	分析函数	LAST_VALUE	返回一组有序值中的最后一个值。
统计函数	分析函数	LEAD	它提供了对表多行的访问，而无需进行自我连接。给定从查询返回的一些列行和光标的位置，LEAD 提供超出该位置的物理偏移量的行的访问。
统计函数	分析函数	LISTAGG	用于列转行。
统计函数	分析函数	NTH_VALUE	返回 analytic_clause 定义的窗口中第 n 行的 measure_expr 值。
统计函数	分析函数	NTILE	将有序数据集划分为 expr 指示的若干桶，并为每一行分配适当的桶号。
统计函数	分析函数	PERCENT_RANK	类似于 CUME_DIST（累积分布）函数。 它的返回值范围为 0~1。任何集合中的第一行的 PERCENT_RANK 函数为 0，返回值为 NUMBER。
统计函数	分析函数	RANK	基于 OVER 子句中的 ORDER BY 表达式确定一组值的排名。
统计函数	分析函数	RATIO_TO_REPORT	计算一个值与一组值之和的比率。
统计函数	分析函数	ROW_NUMBER	为应用它的每一行分配一个唯一的数字。
统计函数	分析函数	STDDEV	用于计算总体标准差。
统计函数	分析函数	STDDEV_POP	计算总体标准差。
统计函数	分析函数	STDDEV_SAMP	计算样本标准差。
统计函数	分析函数	VARIANCE	返回参数指定列的方差。

更多信息

- 数据类型的隐式转换的更多信息，请参阅 数据类型转换 。
- 分析函数中的关键字 OVER，请参阅 窗口函数说明 。

6.2 数字函数

6.2.1 ABS

ABS 函数是返回指定数值表达式的绝对值（正值）的数学函数。ABS 将负值更改为正值，对零或正值没有影响。

语法

```
ABS (numeric_expression)
```

参数

参数	说明
numeric_expression	精确数值或近似数值数据类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

返回类型与参数 numeric_expression 的数据类型相同。

示例

本示例显示了对三个不同数字使用 ABS 函数所得的结果。

执行以下语句：

```
SELECT ABS(-1.0), ABS(0.0), ABS(1.0), ABS(1.666) FROM DUAL;
```

查询结果如下：

```
+-----------+----------+----------+------------+
| ABS(-1.0) | ABS(0.0) | ABS(1.0) | ABS(1.666) |
+-----------+----------+----------+------------+
| 1 | 0 | 1 | 1.666 |
+-----------+----------+----------+------------+
```

6.2.2 ACOS

ACOS 函数返回以弧度表示的角，其余弦为指定的 NUMBER 表达式，也称为反余弦。

语法

```
ACOS (num_expression)
```

参数

参数	值	说明
num_expression	-1.00~1.00	NUMBER 类型或可隐式转换为 NUMBER 类型的表达式 。 仅介于 -1.00 到 1.00 之间的值有效。 对于超出此范围的值，将返回 NULL 且报告域错误。

返回类型

NUMBER 数据类型。

示例

此示例返回指定数量的 ACOS 的值。

执行以下语句：

```
SELECT ACOS(0.3)"acos_test"FROM DUAL;
```

查询结果如下：

```
+-----------+
| acos_test |
+-----------+
|1.26610367 |
+-----------+
```

6.2.3 BITAND

运算符按位进行“与”操作。输入和输出类型均为 NUMBER 数据类型。

语法

```
BITAND (nExpression1, nExpression2)
```

参数

参数	说明
nExpression1, nExpression2	指定按位进行 AND 运算的两个数值。如果 nExpression1 和 nExpression2 为非整数型，那么它们在按位进行 AND 运算之前转换为整数。

返回类型

NUMBER 数据类型。

示例

a 的二进制为 0100；b 的二进制为 0110；比较该二进制，若相对应的位置都为 1，则该位的值为 1，否则值为 0。所以 BITAND(2，3) = 0100 = 2。

执行以下语句：

```
SELECT BITAND(2,3) FROM DUAL;
```

查询结果如下：

```
+-------------+
| BITAND(2,3) |
+-------------+
| 2 |
+-------------+
```

6.2.4 CEIL

CEIL 函数返回值大于等于数值 numeric_expression 的最小整数。

语法

```
CEIL (numeric_expression)
```

参数

参数	说明
numeric_expression	精确数值或近似数值数据类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

返回类型与参数 numeric_expression 的数据类型相同。

示例

此示例显示了对三个不同数字使用 CEIL 函数所得的结果。

执行以下语句：

```
SELECT CEIL(1.2), CEIL(2), CEIL(-12.1) FROM DUAL;
```

查询结果如下：

```
+-----------+---------+-------------+
| CEIL(1.2) | CEIL(2) | CEIL(-12.1) |
+-----------+---------+-------------+
| 2 | 2 | -12 |
+-----------+---------+-------------+
```

6.2.5 EXP

EXP 函数返回 e 的 numeric_expression 次幂（e 为数学常量，e = 2.71828183... ）。

语法

```
EXP (numeric_expression)
```

参数

参数	说明
numeric_expression	精确数值或近似数值数据类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

返回类型与参数 numeric_expression 的数据类型相同。

示例

此示例显示了查询 e 的 4 次幂结果。

执行以下语句：

```
SELECT EXP(4)"e to the 4th power"FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------+
| e to the 4th power |
+-----------------------------------------+
| 54.59815003314423907811026120286087840279 |
+-----------------------------------------+
```

6.2.6 FLOOR

FLOOR 函数返回小于等于数值 numeric_expression 的最大整数。

语法

```
FLOOR (numeric_expression)
```

参数

参数	说明
numeric_expression	精确数值或近似数值数据类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

返回类型与参数 numeric_expression 的数据类型相同。

示例

此示例显示了对三个不同数字使用 FLOOR 函数所得的结果。

执行以下语句：

```
SELECT FLOOR(1.2), FLOOR(2), FLOOR(-12.1) FROM DUAL;
```

查询结果如下：

```
+------------+----------+--------------+
```

```
| FLOOR(1.2) | FLOOR(2) | FLOOR(-12.1) |
+------------+----------+--------------+
| 1 | 2 | -13 |
+------------+----------+--------------+
```

6.2.7 LN

LN 函数返回以 e 为底的 numeric_expression 的对数（e 为数学常量 e = 2.71828183...）。

语法

```
LN (numeric_expression)
```

参数

参数	说明
numeric_expression	精确数值或近似数值数据类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

当参数为 BINARY_FLOAT 数据类型时，返回类型为 BINARY_DOUBLE，其他情况下返回类型与参数 numeric_expression 的数据类型相同。

示例

此示例显示了计算以 e 为底的 4 次对数的结果。

执行以下语句：

```
SELECT LN(4)"Natural log of 4"FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------+
| Natural log of 4 |
+-----------------------------------------+
| 1.38629436111989061883446424291635313615 |
+-----------------------------------------+
```

6.2.8 LOG

LOG 函数返回以 x 为底的 y 的对数。

语法

```
LOG (x,y)
```

参数

参数	说明
x，y	数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。x 和 y 都必须大于 0。

返回类型

当参数为 BINARY_FLOAT 和 BINARY_DOUBLE 数据类型时，返回类型为 BINARY_DOUBLE，其他情况下返回类型为 NUMBER。

示例

此示例显示了查询了以 2 为底的 8 的对数。

```
SELECT LOG(2,8) FROM DUAL;
```

查询结果如下：

```
+----------+
| LOG(2,8) |
+----------+
| 3 |
+----------+
```

6.2.9 MOD

MOD 函数返回 x 除以 y 的余数。

注意：与 REMAINDER 函数的区别为，在用 REMAINDER (x,y) 和 MOD (x,y) 函数在进行运算时，都用了一个公式 result=x-y*(x/y)，区别在于计算 x/y 时的处理方式不同。在 REMAINDER (x,y) 函数中，采用 ROUND(x/y)，而在 MOD (x,y) 函数中采用 FLOOR(x/y)。

语法

```
MOD (x,y)
```

参数

参数	说明
x，y	x 和 y 为数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。

返回类型

返回类型与 数值优先级高 的参数的数据类型相同。

示例

此示例显示了计算 23/8 以及 24/8 的余数的结果。

执行以下语句：

```
SELECT MOD(23,8), MOD(24,8) FROM DUAL;
```

查询结果如下：

```
+-----------+-----------+
| MOD(23,8) | MOD(24,8) |
+-----------+-----------+
| 7 | 0 |
+-----------+-----------+
```

6.2.10 POWER

POWER 函数返回 x 的 y 次幂。

语法

```
POWER (x, y)
```

参数

参数	说明
x，y	x 和 y 为数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。

返回类型

当参数为 BINARY_FLOAT 和 BINARY_DOUBLE 数据类型时，返回类型为 BINARY_DOUBLE，其他情况下返回类型为 NUMBER。

示例

此示例显示了对三组不同数字使用 POWER 函数所得的结果。

执行以下语句：

```
SELECT POWER(2,2), POWER(1.5,0), POWER(20, -1) FROM DUAL;
```

查询结果如下：

```
+------------+--------------+--------------+
| POWER(2,2) | POWER(1.5,0) | POWER(20,-1) |
+------------+--------------+--------------+
| 4 | 1 | .05 |
```

```
+------------+--------------+--------------+
```

6.2.11 REMAINDER

REMAINDER 函数返回 x 除以 y 的余数。

注意：与 MOD 函数的区别为，在用 REMAINDER (x,y) 和 MOD (x,y) 函数在进行运算时，都用了一个公式 result=x-y*(x/y)，区别在于计算 x/y 时的处理方式不同。在 REMAINDER (x,y) 函数中，采用 ROUND(x/y)，而在 MOD (x,y) 函数中采用 FLOOR(x/y)。在REMAINDER 函数中，当 ROUND(x/y) 的参数 x/y 的值的小数部分恰好为 0.5 时，如果 x/y 的值的整数部分为偶数，不向前一位进位，当 x/y 的值的整数部分为奇数，向前一位进位。列如，ROUND(1.5)=2、ROUND(2.5)=2、ROUND(3.5)=4、ROUND(4.5)=4。

语法

```
REMAINDER (x, y)
```

参数

参数	说明
x，y	x 和 y 为数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。

返回类型

返回类型与 数值优先级高 的参数的数据类型相同。

示例

此示例显示了使用 MOD 以及 REMAINDER 函数计算 1.5/1 的余数的结果，请注意两个函数的区别。

执行以下语句：

```
SELECT MOD(1.5,1), REMAINDER(1.5,1) FROM DUAL;
```

查询结果如下：

```
+------------+------------------+
| MOD(1.5,1) | REMAINDER(1.5,1) |
+------------+------------------+
| 0.5 | -0.5 |
+------------+------------------+
```

6.2.12 ROUND

ROUND 函数返回 numeric 四舍五入后的值。

语法

```
ROUND (numeric[,decimal])
```

参数

参数	说明
numeric	数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。
decimal	如果 decimal 大于等于 0 则将 numeric 四舍五入到 decimal 位小数，如果 decimal 小于 0 则四舍五入到小数点向左第 decimal 位。当 decimal 不为整数时，截取 decimal 的整数部分。不指定 decimal 时，将 numeric 四舍五入到整数位。

返回类型

不指定 decimal 时返回类型与参数 numeric 的类型相同，指定 decimal 时，返回类型为 NUMBER 数据类型。

示例

此示例显示了在 decimal 的不同值下四舍五入 5555.6666 的结果。

执行以下语句：

```
SELECT ROUND(5555.6666, 2.1), ROUND(5555.6666, -2.6), ROUND(5555.6666) FROM DUAL;
```

查询结果如下：

```
+---------------------+----------------------+-----------------+
| ROUND(5555.6666,2.1) | ROUND(5555.6666,-2.6) | ROUND(5555.6666) |
+---------------------+----------------------+-----------------+
| 5555.67 | 5600 | 5556 |
+---------------------+----------------------+-----------------+
```

6.2.13 SIGN

SIGN 函数返回数字 n 的符号，大于 0 返回 1，小于 0 返回 -1 ，等于 0 返回 0。

语法

```
SIGN (n)
```

参数

参数	说明
n	精确数值或近似数值数据类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

返回数值 0，1 和 -1。

示例

此示例显示了对三个不同数字使用 SIGN 函数所得的结果。

执行以下语句：

```
SELECT SIGN(100), SIGN(-100), SIGN(0) FROM DUAL;
```

查询结果如下：

```
+-----------+------------+---------+
| SIGN(100) | SIGN(-100) | SIGN(0) |
+-----------+------------+---------+
| 1 | -1 | 0 |
+-----------+------------+---------+
```

6.2.14 SQRT

SQRT 函数返回参数 n 的平方根。

语法

```
SQRT (n)
```

参数

参数	说明
n	数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。n 不能为负数。

返回类型

返回类型与参数 n 的数据类型相同。

示例

此示例显示了对两个不同数字使用 SQRT 函数求平方根的结果.

执行以下语句：

```
SELECT SQRT(64), SQRT(10) FROM DUAL;
```

查询结果如下：

```
+----------+-----------------------------------------+
| SQRT(64) | SQRT(10) |
+----------+-----------------------------------------+
```

```
| 8 | 3.16227766016837933199889354443271853372 |
+----------+------------------------------------------+
```

6.2.15 TRUNC

TRUNC 函数返回参数 numeric 按精度 precision 截取后的值。

语法

```
TRUNC (numeric[,precision])
```

参数

参数	说明
numeric，precision	数值型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）表达式。如果 precision 不为整数则截取 precision 的整数部分。 precision 大于等于 0 时截取 numeric 到 precision 位小数， precision 小于 0 时截取到 numeric 小数点向左第 precision 位，小数前其它数据用 0 表示。precision 的默认值为 0。

返回类型

不指定 precision 时，返回类型与参数 numeric 的数据类型相同。指定 precision 时，返回类型为 NUMBER。

示例

此示例显示了在 precision 的不同值下 5555.66666 的计算结果。

执行以下语句：

```
SELECT TRUNC(5555.66666, 2.1), TRUNC(5555.66666, -2.6), TRUNC(5555.66666) FROM DUAL;
```

查询结果如下：

```
+----------------------+----------------------+--------------------+
| TRUNC(5555.66666,2.1) | TRUNC(5555.66666,-2.6) | TRUNC(5555.033333) |
+----------------------+----------------------+--------------------+
| 5555.66 | 5500 | 5555 |
+----------------------+----------------------+--------------------+
```

6.3 返回字符串的字符串函数

6.3.1 CHR

CHR 函数将 n 转换为等价的一个或多个字符返回，且返回值与当前系统的字符集相关。

语法

```
CHR ( n )
```

参数

参数	取值范围
n	0~4294967295

返回类型

返回值与当前系统的字符集相关。而 OceanBase 支持的字符集是 UTF-8、UTF-16、GBK和GB18030。

示例

十进制（25700）-> 十六进制（0x6464）-> UTF-8编码（dd）执行以下语句：

```
SELECT CHR(25700) AS str FROM DUAL;
```

查询结果如下：

```
+------+
| STR |
+------+
| dd |
+------+
```

十进制（50318）-> 十六进制（0xC48E）-> UTF-8编码（Ď） 执行以下语句：

```
SELECT CHR(50318) AS str FROM DUAL;
```

查询结果如下：

```
+------+
| STR |
+------+
| Ď |
+------+
```

十进制 -> UTF-8编码

```
SELECT CHR(67)||CHR(65)||CHR(84)"Dog"FROM DUAL;
```

查询结果如下：

```
+------+
| Dog |
```

```
+------+
| CAT |
+------+
```

6.3.2 CONCAT

CONCAT 函数可以连接两个字符串。

语法

```
CONCAT(c1,c2)
```

参数

参数	说明
c1	字符串，字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。
c1	字符串，字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。

返回类型

返回与 c1 相同的数据类型。

示例

本示例 CONCAT 函数用来连接字符 '010-' 和字符 '88888888'。执行以下语句

```
SELECT concat('010-','88888888')||'转23'XXXX的电话 FROM DUAL;
```

查询结果如下：

```
+-------------------+
| XXXX的电话 |
+-------------------+
| 010-88888888转23 |
+-------------------+
```

6.3.3 INITCAP

INITCAP 函数返回字符串并将字符串中每个单词的首字母大写，其他字母小写。

语法

```
INITCAP(c1)
```

参数

参数	说明
c1	字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR 或 NVARCHAR2。

返回类型

返回 CHAR 类型数据。

示例

执行以下语句：

```
SELECT initcap('smith abc aBC') upp FROM DUAL;
```

查询结果如下：

```
+------------------+
| UPP |
+------------------+
| Smith Abc Abc |
+------------------+
```

6.3.4 LOWER

LOWER 函数将字符串全部转为小写。

语法

```
LOWER(c1)
```

参数

参数	说明
c1	表示字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。

返回类型

返回与 c1 相同的数据类型 。

示例

执行以下语句：

```
SELECT lower('AaBbCcDd')AaBbCcDd FROM DUAL;
```

查询结果如下：

```
+----------+
| AABBCCDD |
+----------+
| aabbccdd |
+----------+
```

说明：另外， UPPER 将字符串全部转为大写。

6.3.5 LPAD

LPAD 函数的功能是在字符串 c1 的左边用字符串 c2 填充，直到长度为 n 时为止。

语法

```
LPAD(c1,n[,c2])
```

参数

参数	说明
c1	表示字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 。
n	追加后字符总长度，必须是 NUMBER 整数类型或可以隐式转换为 NUMBER 整数类型的类型。
c2	表示追加的字符串，默认为空格。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 。

返回类型

如果 c1 是字符型数据，则返回 VARCHAR2 类型；如果 c1 是国家字符数据类型，则返回 NVARCHAR2 类型；如果 c1 是 LOB 数据类型,，则返回 LOB 类型。

注意：如果 c1 长度大于 n，则返回 c1 左边 n 个字符。如果 c1 长度小于 n，c2 和 c1 连接后大于 n，则返回连接后的右边 n 个字符。

示例

执行以下语句：

```
SELECT lpad('gao',10,'*') FROM DUAL;
```

查询结果如下：

```
+--------------------+
| LPAD('GAO',10,'*') |
+--------------------+
| *******gao |
+--------------------+
```

6.3.6 LTRIM

LTRIM 函数功能是删除左边出现的字符串。

语法

```
LTRIM(c1 [,c2])
```

LTRIM 函数从左端删除 c1 中包含的所有字符 c2。如果未指定 c2，则默认为单个空格。

参数

参数	说明
c1	表示字符串。字符串类型可为： CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 。
c2	表示要删除的字符串，默认为空格。字符串类型可为： CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 。

返回类型

如果 c1 是 CHAR、VARCHAR2 数据类型，则函数返回 VARCHAR2 数据类型。如果 c1 是 NCHAR、NVARCHAR2 数据类型，则函数返回 NVARCHAR2 数据类型。如果 c1 是 LOB 数据类型，则返回的字符串为 LOB 数据类型。

示例

执行以下语句：

```
SELECT LTRIM(' gao qian jing',' ') text FROM DUAL;
```

查询结果如下：

```
+------------------+
| TEXT |
+------------------+
| gao qian jing |
+------------------+
```

您也可以看看 RTRIM 。

6.3.7 REGEXP_REPLACE

REGEXP_REPLACE 函数用于正则表达式替换。

语法

```
REGEXP_REPLACE (source_char, pattern [,replace_string [, position [, occurrence [, match_param ] ] ] ])
```

参数

参数	说明

source_char	用作搜索值的字符表达式。它通常是一种字符列，数据类型可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 。
pattern	是正则表达式 ，它通常是一个文本文字，数据类型可以是 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2。
replace_string	表示替换的字符，可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 类型。
position	是一个正整数类型，指示 OceanBase 从第几个字符开始搜索 source_char 的字符，默认为 1，表示从第一个字符开始搜索 source_char。
occurrence	是一个非负整数，指示替换操作的发生，如果指定 0，则 OceanBase 替换所有匹配项；如果指定正整数 n，则 OceanBase 将替换第 n 次出现，默认为全部都替换掉，如果指定参数 0 也是全部匹配。
match_param	是数据类型 VARCHAR2 或 CHAR 的字符表达式，它允许您更改函数的默认匹配行为。i 表示大小写不敏感；c 表示大小写敏感；n 表示点号；. 表示不匹配换行符号；m 表示多行模式；x 表示忽略空格字符，默认情况下，空格字符会相互匹配。

返回类型

返回结果与 source_char 的数据类型相同。

示例

下面的示例检查字符串，查找两个或多个空格。OceanBase 用一个空间替换两个或多个空间的每一次出现。执行以下语句：

```
SELECT REGEXP_REPLACE('500 OceanBase Parkway, Redwood Shores, CA', '( ){2,}', ' ')"REGEXP_REPLACE"
FROM DUAL;
```

查询结果如下：

```
REGEXP_REPLACE
-------------------------------------
500 OceanBase Parkway, Redwood Shores, CA
```

6.3.8 REPLACE

REPLAC 函数将字符表达式值中，部分相同字符串，替换成新的字符串。

语法

```
REPLACE(c1,c2[,c3])
```

参数

参数	说明
c1	等待替换的字符串 CHAR
c2	搜索需要替换的字符串
c3	替换字符串，默认为空(即删除之意，不是空格)

c1、c2 和 c3 的数据类型是 CHAR 、VARCHAR2、NCHAR、NVARCHAR2 和 CLOB。

返回类型

返回的字符串与 c1 的字符集相同。

如果 c3 缺省或者为 NULL，那么所有有 c1 中出现的 c2 都将被移除。如果 c2 为 NULL，那么结果就是 c1 。如果 c1 是 LOB 数据类型，则函数返回 CLOB 数据类型。如果 c1 不是 LOB 数据类型，则函数返回 VARCHA2 数据类型。

示例

执行以下语句：

```
SELECT replace('he love you','he','i') test FROM DUAL;
```

查询结果如下：

```
+------------+
| TEST |
+------------+
| i love you |
+------------+
```

6.3.9 RPAD

RPAD 函数在字符串 c1 的右边用字符串 c2 填充，直到长度为 n 时为止。

语法

```
RPAD(c1,n[,c2])
```

参数

参数	说明
c1	表示字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。
n	追加后字符总长度，必须是 NUMBER 整数类型或可以隐式转换为 NUMBER 整数类型的类型。
c2	表示追加的字符串，默认为空格。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。

返回类型

如果 c1 是字符型数据，则返回 VARCHAR2 类型；如果 c1 是国家字符数据类型，则返回 NVARCHAR2 类型；如果 c1 是 LOB 数据类型,，则返回 LOB 类型。

说明：

- 如果 c1 长度大于 n，则返回 c1 左边 n 个字符；

- 如果 c1 长度小于 n，c1 和 c2 连接后大于 n，则返回连接后的左边 n 个字符；
- 如果 c1 长度小于n，c1 和 c2 连接后小于 n，则返回 c1 与多个重复 c2 连接（总长度>= n）后的左边 n 个字符。

示例

执行以下语句：

```
SELECT rpad('gao',10,'*a') FROM DUAL;
```

查询结果如下：

```
+---------------------+
| RPAD('GAO',10,'*A') |
+---------------------+
| gao*a*a*a* |
+---------------------+
```

6.3.10 RTRIM

RTRIM 函数删除右边出现的字符串，此函数对于格式化查询的输出非常有用。

语法

```
RTRIM(c1 [,c2])
```

RTRIM 从 c1 中出现的所有字符的右端删除 c2。如果未指定 c2，则默认为单个空格。

参数

参数	说明
c1	表示字符串
c2	表示要删除的字符串，默认为空格

c1、c2 的数据类型可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2 和 CLOB。

返回类型

如果 c1 是 CHAR、VARCHAR2 数据类型，则函数返回 VARCHAR2 数据类型。如果 c1 是 NCHAR、NVARCHAR2 数据类型，则函数返回 NVARCHAR2 数据类型。如果 c1 是 LOB 数据类型，则返回的字符串为 LOB 数据类型。

示例

执行以下语句：

```
SELECT RTRIM('gao qian jingXXXX','X') text FROM DUAL;
```

查询结果如下：

```
+------------------+
| TEXT |
+------------------+
| gao qian jing |
+------------------+
```

6.3.11 SUBSTR

SUBSTR 函数截取子字符串。其中多字节符（汉字、全角符等）按 1 个字符计算。

语法

```
SUBSTR(c1,n1[,n2])
```

参数

参数	说明
c1	需要截取的字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。
n1	截取字符串的开始位置，当 n1 等于 0 或 1 时，都是从第一位开始截取。
n2	要截取的字符串的长度，若不指定 n2，则从第 n1 个字符直到结束的字串。

返回类型

返回与 c1 类型相同的数据。如果 n2 为 0，则将其视为 1。如果 n2 为正，则 OceanBase 从开始算起，从 c1 开始查找第一个字符。如果 n2 为负，则 OceanBase 从 c1 的末尾开始倒数 c1。如果 n3 省略，则 OceanBase 将 c1 的所有字符返回。如果 n3 小于 1，则 OceanBase 返回 NULL。

示例

执行以下语句：

```
SELECT SUBSTR('13088888888',3,8) test FROM DUAL;
```

查询结果如下：

```
+----------+
| TEST |
+----------+
| 08888888 |
+----------+
```

6.3.12 TRANSLATE

TRANSLATE 函数将字符表达式值中，指定字符替换为新字符。多字节符（汉字、全角符等），按 1 个字符计算。

语法

```
TRANSLATE(c1,c2,c3)
```

参数

参数	说明
c1	希望被替换的字符或变量。
c2	查询原始的字符集。
c3	替换新的字符集，将 c2 对应顺序字符，替换为 c3 对应顺序字符 。

说明：

- c1、c2 和 c3 的数据类型可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 。
- 如果 c3 长度大于 c2，则 c3 长出后面的字符无效；
- 如果 c3 长度小于 c2，则 c2 长出后面的字符均替换为空（删除）；
- 如果 c3 长度为 0，则返回空字符串；
- 如果 c2 里字符重复，按首次位置为替换依据。

返回类型

返回 CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 类型字符。

示例

执行以下语句：

```
SELECT TRANSLATE('he love you','he','i') FROM DUAL;
```

查询结果如下：

```
+------------------+
| TRANSLATE |
+------------------+
| i love you |
+------------------+
```

6.3.13 TRIM

TRIM 函数用来删除一个字符串的开头或结尾（或两者）的字符。

语法

```
TRIM([ { { LEADING | TRAILING | BOTH } [ trim_character ] | trim_character } FROM ] trim_source)
```

参数

参数	说明
LEADING	开头字符。
TRAILING	结尾字符。
BOTH	开头和结尾字符。
trim_character	删除的字符。
trim_source	修剪源。

trim_char 和 trim_source 都可以是 VARCHAR2 或任何可以隐式转换为 VARCHAR2 的数据类型。如果指定 LEADING，则 OceanBase 将删除所有与前导字符相等的 trim_character。如果指定 TRAILING，则 OceanBase 将删除所有与结尾字符相等的 trim_character。如果您指定 BOTH 或三个都不指定，则 OceanBase 删除与前导和结尾字符相等的 trim_character。如果未指定 trim_character，则默认值为空白。如果仅指定 trim_source，则 OceanBase 删除前导和尾随空格。如果函数返回的值数据类型为 VARCHAR2，则该值的最大长度为 trim_source 。

返回类型

如果 trim_source 为 CHAR、VARCHAR2 数据类型，则函数返回 VARCHAR2 数据类型。如果 trim_source 为 NCHAR、NVARCHAR2 数据类型，则函数返回 NVARCHAR2 数据类型。如果 trim_source 为 CLOB 数据类型，则函数返回 CLOB 数据类型。如果 trim_source 或 trim_character 为 NULL，则 TRIM 函数返回 NULL。

示例

执行以下语句：

```
SELECT TRIM('X' from 'XXXgao qian jingXXXX'),TRIM('X' from 'XXXgaoXXjingXXXX') text FROM DUAL;
```

查询结果如下：

```
+----------------------------------+-----------+
| TRIM('X'FROM'XXXGAOQIANJINGXXXX') | TEXT |
+----------------------------------+-----------+
| gao qian jing | gaoXXjing |
+----------------------------------+-----------+
```

6.3.14 UPPER

UPPER 函数将字符串全部转为大写。

语法

```
UPPER(c1)
```

参数

参数	说明
c1	字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。

返回类型

返回与 c1 相同的数据类型。

示例

执行以下语句：

```
SELECT UPPER('AaBbCcDd') upper FROM DUAL;
```

查询结果如下：

```
+----------+
| UPPER |
+----------+
| AABBCCDD |
+----------+
```

说明：另外， LOWER 函数可以将字符串全部转为小写。

6.4 返回数字的字符串函数

6.4.1 ASCII

ASCII 函数返回字符表达式最左端字符的 ASCII 码值。

语法

```
ASCII(x)
```

参数

参数	说明
x	CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的表达式。

返回类型

NUMBER 数据类型。

示例

此示例使用 ASCII 函数返回 **A**、**a**、**中** 和空格的 ASCII 码值。

```
SELECT ASCII('A') A, ASCII('a') a, ASCII(' ') space,ASCII('中') hz FROM DUAL;
```

结果返回：

```
+----+----+-------+-----+
| A | a | space | hz |
+----+----+-------+-----+
| 65 | 97 | 32 | 228 |
+----+----+-------+-----+
```

6.4.2 INSTR

INSTR 函数在一个字符串中搜索指定的字符，返回发现指定的字符的位置。

注意：多字节符（汉字、全角符等），按 1 个字符计算。

语法

```
INSTR(c1,c2[,i[,j]])
```

参数

参数	说明
c1	被搜索的字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。
c2	希望搜索的字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB。
i	搜索的开始位置，默认值为 1。当值小于 0 时，搜索从反方向开始，但是仍返回被搜索到的字符的正序位置。
j	第 j 次出现的位置，默认值为1。

返回类型

NUMBER 数据类型。

示例

此示例展示了字符 **某** 在字符串 **重庆某软件公司** 中第一次出现的位置。

```
SELECT INSTR ('重庆某软件公司','某',1,1) instring FROM DUAL;
```

返回结果：

```
+----------+
| instring |
+----------+
| 3 |
+----------+
```

此示例中， **instring1** 应返回正向搜索 **ce** 时在字符串中第二次出现的位置，**instring2** 应返回反向搜索 **ce** 时在字符串中第二次出现的位置：

```
SELECT INSTR ('oceanbase pratice','ce',1,2) instring1，INSTR ('oceanbase pratice','ce',-1,2) instring2 FROM DUAL;
```

返回结果，正向搜索时 **ce** 第二次出现在第 16 位，反向搜索时 **ce** 第二次出现在第二位：

```
+----------+------------+
| instring1 | instring2 |
+----------+------------+
| 16 | 2 |
+----------+------------+
```

6.4.3 LENGTH

LENGTH 函数返回字符串的长度。

注意：多字节符（汉字、全角符等），按 1 个字符计算。

语法

```
LENGTH(c1)
```

参数

参数	说明
c1	CHAR、VARCHAR2、NCHAR、NVARCHAR2 或 CLOB 数据类型的字符串。

返回类型

NUMBER 数据类型。

示例

此示例展示了字符串 **测试、北京市海淀区** 和 **北京TO_CHAR** 的长度。

```
SELECT LENGTH ('测试'), LENGTH('北京市海锭区'), LENGTH('北京TO_CHAR') FROM DUAL;
```

返回结果：

```
+-----------------+----------------------------+-----------------------+
| LENGTH('测试')| LENGTH('北京市海锭区') | LENGTH('北京TO_CHAR') |
+-----------------+----------------------------+-----------------------+
| 2 | 6 | 9 |
+-----------------+----------------------------+-----------------------+
```

6.5 日期时间函数

6.5.1 ADD_MONTHS

ADD_MONTHS 函数返回在日期 date 基础上 n 个月后的日期值，如果 n 的值为负数则返回日期 date 基础上 n 个月前的日期值（date 减去 n 个月）。

注意：由于每个月的天数不同，当 date 是一个月中的最后一天时，函数返回计算后该月的最后一天。例如，用 ADD_MONTHS 计算 2020 年 3 月 31 日一个月前的日期，返回 2020 年 2 月 29 日。

语法

```
ADD_MONTHS (date,n)
```

参数

参数	说明
date	DATE 数据类型。
n	NUMBER 数据类型。

返回类型

DATE 数据类型。

示例

示例 1： 以下示例查询了 3 个月后的日期。

执行以下语句：

```
SELECT SYSDATE, ADD_MONTHS(SYSDATE,3) FROM DUAL;
```

查询结果如下：

```
+---------------------+----------------------+
| SYSDATE | ADD_MONTHS(SYSDATE,3) |
+---------------------+----------------------+
| 2020-03-26 12:21:40 | 2020-06-26 12:21:40 |
+---------------------+----------------------+
```

示例 2： 以下示例查询了 3 个月前的日期。

执行以下语句：

```
SELECT SYSDATE, ADD_MONTHS(SYSDATE,-3) FROM DUAL;
```

查询结果如下：

```
+--------------------+------------------------+
| SYSDATE | ADD_MONTHS(SYSDATE,-3) |
+--------------------+------------------------+
| 2020-03-26 12:21:04 | 2019-12-26 12:21:04 |
+--------------------+------------------------+
```

6.5.2 CURRENT_DATE

CURRENT_DATE 函数返回当前会话时区中的当前日期。

语法

```
CURRENT_DATE
```

参数

无参数。

返回类型

DATE 数据类型。

示例

此示例显示了不同会话时区使用 CURRENT_DATE 函数所得的结果。

设置当前时区为 GMT-5 时区：

```
ALTER SESSION SET TIME_ZONE = '-05:00';
```

执行以下语句调用函数：

```
SELECT CURRENT_DATE FROM DUAL;
```

查询结果如下：

```
+---------------------+
| CURRENT_DATE |
```

```
+---------------------+
| 2020-03-08 01:40:11 |
+---------------------+
```

切换当前时区至 GMT+8 时区：

```
ALTER SESSION SET TIME_ZONE = '+08:00';
```

执行以下语句调用函数：

```
SELECT CURRENT_DATE FROM DUAL;
```

查询结果如下：

```
+---------------------+
| CURRENT_DATE |
+---------------------+
| 2020-03-08 14:40:11 |
+---------------------+
```

6.5.3 CURRENT_TIMESTAMP

CURRENT_TIMESTAMP 函数返回 TIMESTAMP WITH TIME ZONE 数据类型的当前会话时区中的当前日期，返回值中包含当前的时区信息。

语法

```
CURRENT_TIMESTAMP (precision )
```

参数

参数	说明
precision	表示秒小数位的精度，默认值为 6，取值范围 0~9。

返回类型

包含当前的时区信息的 TIMESTAMP WITH TIME ZONE 数据类型。

示例

此示例显示了不同会话时区使用 CURRENT_TIMESTAMP 函数所得的结果。

设置当前时区至 GMT-5 时区：

```
ALTER SESSION SET TIME_ZONE = '-05:00';
```

执行以下语句调用函数：

```
SELECT CURRENT_TIMESTAMP FROM DUAL;
```

查询结果如下：

```
+-----------------------------------+
| CURRENT_TIMESTAMP |
+-----------------------------------+
| 2020-03-08 01:49:31.219066 -05:00 |
+-----------------------------------+
```

切换当前时区至 GMT+8 时区，且调整秒的小数位精度为 3：

```
ALTER SESSION SET TIME_ZONE = '+08:00';
```

执行以下语句调用函数：

```
SELECT CURRENT_TIMESTAMP(3) FROM DUAL;
```

查询结果如下：

```
+--------------------------------+
| CURRENT_TIMESTAMP(3) |
+--------------------------------+
| 2020-03-08 14:50:32.499 +08:00 |
+--------------------------------+
```

6.5.4 DBTIMEZONE

DBTIMEZONE 函数返回当前数据库实例的时区，在 OceanBase 中数据库时区恒为+00:00，且不支持修改。

语法

```
DBTIMEZONE
```

参数

无参数。

返回类型

VARCHAR2 数据类型。

示例

执行以下语句：

```
SELECT DBTIMEZONE FROM DUAL;
```

查询结果如下：

```
+------------+
| DBTIMEZONE |
+------------+
| +00:00 |
+------------+
```

6.5.5 EXTRACT（datetime）

EXTRACT(datetime) 函数是从指定的时间字段或表达式中抽取年、月、日、时、分、秒等元素。

语法

```
EXTRACT (fields FROM datetime)
```

参数

参数	说明
fields	要抽取的元素的名称： YEAR、MONTH、DAY、HOUR、MINUTE、SECOND、TIMEZONE_HOUR、TIMEZONE_MINUTE、TIMEZONE_MINUTE、TIMEZONE_REGION、TIMEZONE_ABBR。
datetime	DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL YEAR TO MONTH、INTERVAL DAY TO SECOND 等数据类型的值。

返回类型

抽取元素 TIMEZONE_REGION、TIMEZONE_ABBR 时，返回值的数据类型为 VARCHAR2。抽取其他元素时，返回值的数据类型为 NUMBER。

示例

执行以下语句：

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-2-16 2:38:40 ') 小时,
EXTRACT(MINUTE FROM TIMESTAMP '2001-2-16 2:38:40 ') 分钟,
EXTRACT(SECOND FROM TIMESTAMP '2001-2-16 2:38:40 ') 秒,
EXTRACT(DAY FROM TIMESTAMP '2001-2-16 2:38:40 ') 日,
EXTRACT(MONTH FROM TIMESTAMP '2001-2-16 2:38:40 ') 月,
EXTRACT(YEAR FROM TIMESTAMP '2001-2-16 2:38:40 ') 年
FROM DUAL;
```

查询结果如下：

```
+--------+--------+------+------+------+------+
```

```
| 小时 | 分钟 | 秒 | 日 | 月 | 年 |
+--------+--------+------+------+------+------+
| 2 | 38 | 40 | 16 | 2 | 2001 |
+--------+--------+------+------+------+------+
```

6.5.6 FROM_TZ

FROM_TZ 函数将一个 TIMSTAMP 数据类型的值和时区信息拼成一个 TIMESTAMP WITH TIME ZONE 数据类型的时间值。

语法

```
FROM_TZ (timestamp_value,time_zone_value)
```

参数

参数	说明
timestamp_value	TIMSTAMP 数据类型的时间值。
time_zone_value	时区信息。

返回类型

TIMESTAMP WITH TIME ZONE 数据类型。

示例

执行以下语句：

```
SELECT FROM_TZ(TIMESTAMP '2020-03-28 08:00:00', '-03:00') FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------------+
| FROM_TZ(TIMESTAMP'2020-03-2808:00:00','-03:00') |
+-----------------------------------------------+
| 2020-03-28 08:00:00.000000000 -03:00 |
+-----------------------------------------------+
```

6.5.7 LAST_DAY

LAST_DAY 函数返回日期 date 所在月份的最后一天的日期。

语法

```
LAST_DAY (date)
```

参数

参数	说明
date	所有包含日期信息的数据类型（DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE）的值。

返回类型

DATE 数据类型。

示例

本示例显示了当前日期与本月的最后一天日期以及本月剩余的天数。

执行以下语句：

```
SELECT SYSDATE, LAST_DAY(SYSDATE)Last, LAST_DAY(SYSDATE)-SYSDATE"Left"FROM DUAL;
```

查询结果如下：

```
+--------------------+--------------------+------+
| SYSDATE | LAST | Left |
+--------------------+--------------------+------+
| 2020-03-08 15:23:33 | 2020-03-31 15:23:33 | 23 |
+--------------------+--------------------+------+
```

6.5.8 LOCALTIMESTAMP

LOCALTIMESTAMP 函数返回当前会话时区中的当前日期，返回 TIMESTAMP 数据类型的值。与函数 CURRENT_TIMESTAMP 的区别是，CURRENT_TIMESTAMP 函数返回 TIMESTAMP WITH TIME ZONE 数据类型的值。

语法

```
LOCALTIMESTAMP (timestamp_precision )
```

参数

参数	说明
timestamp_precision	表示秒的小数位的精度，默认值为 6，取值范围 0~9。

返回类型

TIMESTAMP 数据类型。

示例

此示例显示了不同会话时区使用 LOCALTIMESTAMP 函数所得的结果。

设置当前会话时区至 GMT-5 时区：

```
ALTER SESSION SET TIME_ZONE = '-05:00';
```

执行以下语句调用函数：

```
SELECT LOCALTIMESTAMP FROM DUAL;
```

查询结果如下：

```
+---------------------------+
| LOCALTIMESTAMP |
+---------------------------+
| 2020-03-08 02:30:20.062104 |
+---------------------------+
```

切换当前会话时区至 GMT+8 时区，且调整秒的小数位精度为 3：

```
ALTER SESSION SET TIME_ZONE = '+08:00';
```

执行以下语句调用函数：

```
SELECT LOCALTIMESTAMP(3) FROM DUAL;
```

查询结果如下：

```
+-------------------------+
| LOCALTIMESTAMP(3) |
+-------------------------+
| 2020-03-08 15:30:54.500 |
+-------------------------+
```

6.5.9 MONTHS_BETWEEN

MONTHS_BETWEEN 函数是返回返回参数 date1 到 date2 之间的月数。

语法

```
MONTHS_BETWEEN (date1，date2)
```

参数

参数	说明

date1	DATE 数据类型的值。
date2	DATE 数据类型的值。

返回类型

NUMBER 数据类型。如果 date1>date2，返回正数；如果 date1<date2，则返回负数。

示例

以下示例展示了当前时间和指定时间值之间的月数：

```
SELECT SYSDATE,
MONTHS_BETWEEN(SYSDATE,TO_DATE('2006-01-01','YYYY-MM-DD')),
MONTHS_BETWEEN(SYSDATE,TO_DATE('2022-01-01','YYYY-MM-DD'))
FROM DUAL;
```

查询结果如下：

```
+--------------------+-----------------------------------------------------------+-----------------------------------------------------------+
| SYSDATE | MONTHS_BETWEEN(SYSDATE,TO_DATE('2006-01-01','YYYY-MM-DD')) |
MONTHS_BETWEEN(SYSDATE,TO_DATE('2022-01-01','YYYY-MM-DD')) |
+--------------------+-----------------------------------------------------------+-----------------------------------------------------------+
| 2020-03-08 15:38:35 | 170.246832063918757467144563918757467145 | -
21.753167936081242532855436081242532855544 |
+--------------------+-----------------------------------------------------------+-----------------------------------------------------------+
```

6.5.10 NEXT_DAY

NEXT_DAY 函数是返回日期 d1 的下一周中 c1（星期值）所在的日期值。

语法

```
NEXT_DAY (d1[,c1])
```

参数

参数	说明
d1	DATE 数据类型的值。
c1	星期值：MONDAY、TUESDAY、WEDNESDAY、THURSDAY、FRIDAY、SATURDAY 和 SUNDAY。

返回类型

DATE 数据类型。

示例

以下示例展示了下周每个星期值对应的日期值：

```
SELECT SYSDATE 当时日期,
NEXT_DAY(SYSDATE,'MONDAY') 下周星期一,
NEXT_DAY(SYSDATE,'TUESDAY') 下周星期二,
NEXT_DAY(SYSDATE,'WEDNESDAY') 下周星期三,
NEXT_DAY(SYSDATE,'THURSDAY') 下周星期四,
NEXT_DAY(SYSDATE,'FRIDAY') 下周星期五,
NEXT_DAY(SYSDATE,'SATURDAY') 下周星期六,
NEXT_DAY(SYSDATE,'SUNDAY') 下周星期日
FROM DUAL;
```

查询结果如下：

```
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
| 当时日期 | 下周星期一 | 下周星期二 | 下周星期三 | 下周星期四 | 下周星期五 | 下周星期六 | 下周星期日 |
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
| 2020-03-08 15:47:57 | 2020-03-09 15:47:57 | 2020-03-10 15:47:57 | 2020-03-11 15:47:57 | 2020-03-12 15:47:57 | 2020-03-13 15:47:57 | 2020-03-14 15:47:57 | 2020-03-15 15:47:57 |
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
```

6.5.11 NUMTODSINTERVAL

NUMTODSINTERVAL 函数是把参数 n 转为以参数 interval_unit 为单位的 INTERVAL DAY TO SECOND 数据类型的值。

语法

```
NUMTODSINTERVAL (n,interval_unit)
```

参数

参数	说明
n	NUMBER 数据类型的值。
interval_unit	单位值：DAY、HOUR、MINUTE、MINUTE。

返回类型

INTERVAL DAY TO SECOND 数据类型。

示例

以下示例展示了当前日期 3 小时后的日期时间值：

```
SELECT SYSDATE,SYSDATE+NUMTODSINTERVAL(3,'HOUR') AS RES FROM DUAL;
```

查询结果如下：

```
+---------------------+---------------------+
| SYSDATE | RES |
+---------------------+---------------------+
| 2020-03-08 16:01:40 | 2020-03-08 19:01:40 |
+---------------------+---------------------+
```

6.5.12 NUMTOYMINTERVAL

NUMTOYMINTERVAL 函数是把参数 n 转为以 interval_unit 为单位的 INTERVAL YEAR TO MONTH 数据类型的值。

语法

```
NUMTOYMINTERVAL (n,interval_unit)
```

参数

参数	说明
n	NUMBER 数据类型的值。
interval_unit	单位值：YEAR、MONTH。

返回类型

INTERVAL YEAR TO MONTH 数据类型。

示例

以下示例展示了当前日期 3 年后的日期时间值：

```
SELECT SYSDATE,SYSDATE+NUMTOYMINTERVAL(3,'YEAR') AS RES FROM DUAL;
```

查询结果如下：

```
+---------------------+---------------------+
| SYSDATE | RES |
+---------------------+---------------------+
| 2020-03-08 16:03:58 | 2023-03-08 16:03:58 |
+---------------------+---------------------+
```

6.5.13 ROUND (date)

ROUND(date) 函数返回以参数 fmt 为单位距离的离指定日期 date 最近的日期时间值。

语法

```
ROUND (date，[fmt])
```

参数

参数	说明
date	所有包含日期的数据类型的值：DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE。
fmt	指定了函数返回值与 date 的距离单位，以下表格列举了该参数的可取值，大小写不敏感。

fmt参数表	说明
j	默认值，最近0点日期。
day、dy、d	返回离指定日期最近的星期日。
month、mon、mm、rm	返回离指定日期最近的月的第一天日期。
q	返回离指定日期最近的季的日期。
syear、year、yyyy、yyy、yy、y	多个 y 表示不同的精度，返回离指定日期最近的年的第一个日期。
cc、scc	返回离指定日期最近的世纪的初日期。

返回类型

DATE 数据类型。

示例

执行以下语句：

```
SELECT SYSDATE 当时日期,
ROUND(SYSDATE) 最近0点日期,
ROUND(SYSDATE,'DAY') 最近星期日,
ROUND(SYSDATE,'MONTH') 最近月初,
ROUND(SYSDATE,'Q') 最近季初日期,
ROUND(SYSDATE,'YEAR') 最近年初日期
FROM DUAL;
```

查询结果如下：

```
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 当时日期 | 最近0点日期 | 最近星期日 | 最近月初 | 最近季初日期 | 最近年初日期 |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
| 2020-03-08 20:24:53 | 2020-03-09 00:00:00 | 2020-03-08 00:00:00 | 2020-03-01 00:00:00 | 2020-04-01 00:00:00 | 2020-01-01 00:00:00 |
+---------------------+---------------------+---------------------+---------------------+---------------------+---------------------+
```

6.5.14 SESSIONTIMEZONE

SESSIONTIMEZONE 函数是返回当前会话时区。

语法

```
SESSIONTIMEZONE
```

参数

无参数。

返回类型

VARCHAR2 数据类型。

示例

以下示例展示了数据库时区和当前会话时区：

```
SELECT DBTIMEZONE,SESSIONTIMEZONE FROM DUAL;
```

查询结果如下：

```
+------------+-----------------+
| DBTIMEZONE | SESSIONTIMEZONE |
+------------+-----------------+
| +00:00 | +08:00 |
+------------+-----------------+
```

可以通过 ALTER SESSION 语句修改当前会话时区，数据库时区不可修改：

```
ALTER SESSION SET TIME_ZONE = '+05:00';
```

执行以下语句查询修改后的当前会话时区：

```
SELECT DBTIMEZONE,SESSIONTIMEZONE FROM DUAL;
```

查询结果如下：

```
+------------+-----------------+
| DBTIMEZONE | SESSIONTIMEZONE |
+------------+-----------------+
| +00:00 | +05:00 |
+------------+-----------------+
```

6.5.15 SYSDATE

SYSDATE 函数返回当前日期。

注意：该函数不依赖于当前会话时区，而是依赖于数据库宿主机的 Linux 操作系统的时区。

语法

```
SYSDATE
```

参数

无参数。

返回类型

DATE 数据类型。

示例

以下示例将当前时间按指定格式输出：

```
SELECT TO_CHAR
(SYSDATE, 'MM-DD-YYYY HH24:MI:SS')"NOW"
FROM DUAL;
```

查询结果如下：

```
+---------------------+
| NOW |
+---------------------+
| 03-08-2020 20:44:04 |
+---------------------+
```

6.5.16 SYSTIMESTAMP

SYSTIMESTAMP 函数返回系统当前日期，返回值的秒的小数位包含 6 位精度，且包含当前时区信息。

注意：该函数不依赖于当前会话时区，而是依赖于数据库宿主机的 Linux 操作系统的时区。

语法

```
SYSTIMESTAMP
```

参数

无参数。

返回类型

TIMESTAMP WITH TIME ZONE 数据类型。

示例

执行以下语句：

```
SELECT SYSTIMESTAMP FROM DUAL;
```

查询结果如下：

```
+-----------------------------------+
| SYSTIMESTAMP |
+-----------------------------------+
| 2020-03-08 20:47:08.254086 +08:00 |
+-----------------------------------+
```

6.5.17 SYS_EXTRACT_UTC

SYS_EXTRACT_UTC 函数是返回与指定时间相对应的标准 UTC 时间。

语法

```
SYS_EXTRACT_UTC (datetime_with_timezone)
```

注意：UTC（Universal Time Coordinated）是通用协调时间。UTC 与格林尼治平均时（GMT, Greenwich Mean Time）一样，都与英国伦敦的本地时相同。

参数

参数	说明
datetime_with_timezone	TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE 数据类型的值。

返回类型

TIMESTAMP 数据类型。

示例

执行以下语句：

```
SELECT SYS_EXTRACT_UTC(TIMESTAMP '2020-03-28 11:30:00.00 -08:00')
FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------------------+
| SYS_EXTRACT_UTC(TIMESTAMP'2020-03-2811:30:00.00-08:00') |
+-----------------------------------------------------+
```

```
| 2020-03-28 19:30:00.000000000 |
+-------------------------------------------------------+
```

6.5.18 TO_CHAR (datetime)

TO_CHAR 函数将 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 等数据类型的值按照参数 fmt 指定的格式转换为 VARCHAR2 数据类型的值。 如果不指定参数 fmt，则参数 datetime 的值将按如下格式转换为 VARCHAR2 数据类型：

- DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE 的值被转换为数据库中日期时间值的默认格式。您可在数据类型章节中查看各日期时间类型的默认格式。
- INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 数据类型的值转换为数字格式的间隔值。

语法

```
TO_CHAR( datetime [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
datetime	DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 等数据类型的值。
fmt	输出格式参数，详细格式信息请参见 日期时间格式化元素表 。
nlsparam	用来控制返回的月份和日份所使用的语言。

返回类型

VARCHAR2 数据类型。

示例

示例 1： 以下语句通过 TO_CHAR 函数返回系统当前日期，并且将日期时间值转换为了 **DS DL** 格式：

```
SELECT TO_CHAR(SYSDATE,'DS DL') FROM DUAL;
```

查询结果如下：

```
+-----------------------------------+
| TO_CHAR(SYSDATE,'DSDL') |
+-----------------------------------+
| 03/08/2020 Sunday, March 08, 2020 |
+-----------------------------------+
```

示例 2：以下语句将间隔值转化为指定格式，并且设置了返回语言为 **AMERICAN**：

```
SELECT TO_CHAR(interval'1' year, 'YY-DD', 'nls_language = AMERICAN') FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------------------+
| TO_CHAR(INTERVAL'1'YEAR,'YY-DD','NLS_LANGUAGE=AMERICAN') |
+-----------------------------------------------------+
| +01-00 |
+-----------------------------------------------------+
```

6.5.19 TO_DSINTERVAL

TO_DSINTERVAL 函数将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL DAY TO SECOND 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。

语法

```
TO_DSINTERVAL (days hours：minutes：seconds[.frac_secs])
```

参数

参数	说明
days hours：minutes：seconds[.frac_secs]	符合该参数格式的 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串。

返回类型

INTERVAL DAY TO SECOND 数据类型。

示例

以下示例返回了当前时间 100 天后的日期时间值：

```
SELECT SYSDATE, SYSDATE+TO_DSINTERVAL('100 00:00:00') FROM DUAL;
```

查询结果如下：

```
+--------------------+-----------------------------------+
| SYSDATE | SYSDATE+TO_DSINTERVAL('10000:00:00') |
+--------------------+-----------------------------------+
| 2020-03-26 12:40:39 | 2020-07-04 12:40:39 |
+--------------------+-----------------------------------+
```

6.5.20 TO_TIMESTAMP

TO_TIMESTAMP 函数将字符串转换为 TIMESTAMP 数据类型。

语法

```
TO_TIMESTAMP (char，[fmt],['nlsparam'])
```

参数

参数	说明
char	CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串。
fmt	指定返回值的格式。详细格式信息请参见 日期时间格式化元素表 。
nlsparam	用来控制返回的月份和日份所使用的语言。

返回类型

TIMESTAMP 数据类型。

示例

执行以下语句：

```
SELECT TO_TIMESTAMP ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF')
FROM DUAL;
```

查询结果如下：

```
+---------------------------------------------------------------+
| TO_TIMESTAMP('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF') |
+---------------------------------------------------------------+
| 2002-09-10 14:10:10.123000000 |
+---------------------------------------------------------------+
```

6.5.21 TO_TIMESTAMP_TZ

TO_TIMESTAMP_TZ 函数将字符串转换为 TIMESTAMP WITH TIME ZONE 数据类型，包含时区信息。

语法

```
TO_TIMESTAMP_TZ (char，[fmt],['nlsparam'])
```

参数

参数	说明
char	CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串。
fmt	指定输出格式，详细格式信息请参见 日期时间格式化元素表 。
nlsparam	用来控制返回的月份和日份所使用的语言。

返回类型

TIMESTAMP WITH TIME ZONE 数据类型。

示例

执行以下语句：

```
SELECT TO_TIMESTAMP_TZ ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF') FROM DUAL;
```

查询结果如下：

```
+---------------------------------------------------------------------+
| TO_TIMESTAMP_TZ('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF') |
+---------------------------------------------------------------------+
| 2002-09-10 14:10:10.123000000 +08:00 |
+---------------------------------------------------------------------+
```

6.5.22 TO_YMINTERVAL

TO_YMINTERVAL 函数将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL YEAR TO MONTH 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。

语法

```
TO_YMINTERVAL (years-months)
```

参数

参数	说明
years-months	符合该参数格式的 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串。

返回类型

INTERVAL YEAR TO MONTH 数据类型。

示例

以下示例展示了返回当前时间 1 年 2 个月后的时间日期值：

```
SELECT SYSDATE,SYSDATE+TO_YMINTERVAL('01-02') FROM DUAL;
```

查询结果如下：

```
+--------------------+-------------------------------+
| SYSDATE | SYSDATE+TO_YMINTERVAL('01-02') |
+--------------------+-------------------------------+
| 2020-03-08 22:32:01 | 2021-05-08 22:32:01 |
```

```
+--------------------+--------------------------------+
```

6.5.23 TRUNC (date)

TRUNC 函数返回以参数 fmt 为单位距离的离指定日期 date 最近的日期时间值，并且返回的日期值在 date 之前。

注意：与函数 ROUND 的区别为，TRUNC 返回的值必须是在 date 之前的离 date 最近的日期，ROUND 可以是 date 之前也可以是 date 之后的离它最近的日期值。

语法

```
TRUNC (date,[fmt])
```

参数

参数	说明
date	DATE 数据类型。
fmt	指定了函数返回值与 date 的距离单位，以下表格列举了该参数的可取值，大小写不敏感。

fmt参数表	说明
j	默认值，最近 0 点日期。
day、dy、d	返回离指定日期最近的星期日。
month、mon、mm、rm	返回离指定日期最近的月的第一天日期。
q	返回离指定日期最近的季的日期。
yyyy、yyy、yy、y	多个 y 表示不同的精度，返回离指定日期最近的年的第一个日期。
cc、scc	返回离指定日期最近的世纪的初日期。

返回类型

DATE 数据类型。

示例

以下示例展示了用 TRUNC 计算距离 **SYSDATE** 最近的符合要求的日期值：

```
SELECT SYSDATE 当时日期,
TRUNC(SYSDATE) 今天日期,
TRUNC(SYSDATE,'DAY') 本周星期日,
TRUNC(SYSDATE,'MONTH') 本月初,
TRUNC(SYSDATE,'Q') 本季初日期,
TRUNC(SYSDATE,'YEAR') 本年初日期 FROM DUAL;
```

查询结果如下：

```
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
| 当时日期 | 今天日期 | 本周星期日 | 本月初 | 本季初日期 | 本年初日期 |
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
| 2020-03-08 22:41:46 | 2020-03-08 00:00:00 | 2020-03-08 00:00:00 | 2020-03-01 00:00:00 | 2020-01-01 00:00:00 |
2020-01-01 00:00:00 |
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
```

以下示例展示了在相同日期下用 ROUND 计算符合要求的最近日期的结果：

```
SELECT SYSDATE 当时日期,
ROUND(SYSDATE) 最近0点日期,
ROUND(SYSDATE,'DAY') 最近星期日,
ROUND(SYSDATE,'MONTH') 最近月初,
ROUND(SYSDATE,'Q') 最近季初日期,
ROUND(SYSDATE,'YEAR') 最近年初日期 FROM DUAL;
```

查询结果如下：

```
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
| 当时日期 | 最近0点日期 | 最近星期日 | 最近月初 | 最近季初日期 | 最近年初日期 |
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
| 2020-03-08 22:41:02 | 2020-03-09 00:00:00 | 2020-03-08 00:00:00 | 2020-03-01 00:00:00 | 2020-04-01 00:00:00 |
2020-01-01 00:00:00 |
+--------------------+--------------------+--------------------+--------------------+--------------------+--------------------+
```

6.5.24 TZ_OFFSET

TZ_OFFSET 函数返回时区 n 的时区偏移量。时区偏移量是指与格林尼治标准时间 GMT 的差（小时和分钟）。

语法

```
TZ_OFFSET (n)
```

参数

参数	说明
n	时区区域名称。

返回类型

VARCHAR2 数据类型。

示例

以下示例返回了当前会话时区 SESSIONTIMEZONE 、数据库时区 DBTIMEZONE 和 US/Eastern 所在时区的时区偏移量：

```
SELECT TZ_OFFSET(SESSIONTIMEZONE),TZ_OFFSET(DBTIMEZONE),TZ_OFFSET('US/Eastern') FROM DUAL;
```

查询结果如下：

```
+---------------------------+----------------------+------------------------+
| TZ_OFFSET(SESSIONTIMEZONE) | TZ_OFFSET(DBTIMEZONE) | TZ_OFFSET('US/EASTERN') |
+---------------------------+----------------------+------------------------+
| +08:00 | +00:00 | -04:00 |
+---------------------------+----------------------+------------------------+
```

6.6 通用比较函数

6.6.1 GREATEST

GREATEST 函数返回一个或多个表达式列表中的最大值。 数据库使用第一个参数 expr 来确定返回类型。 如果其余参数的数据类型和第一个参数 expr 的数据类型不同，则 OceanBase 将第一个参数 expr 之后的每个参数隐式转换为比较之前的第一个 expr 的数据类型。

语法

```
GREATEST(expr [, expr ]...)
```

参数

参数	说明
expr	一个表达式或表达式列表。数据类型可以是：NUMBER、FLOAT、BINARY_FLOAT、BINARY_DOUBLE、CHAR、VARCHAR2、NCHAR、NVARCHAR2、或 CLOB。

返回类型

如果第一个参数 expr 的数据类型为 NUMBER、FLOAT、BINARY_FLOAT或 BINARY_DOUBLE，则返回与第一个参数 expr 相同的数据类型。如果第一个参数 expr 的数据类型为 CHAR、VARCHAR2 或 CLOB，则返回 VARCHAR2 类型。如果第一个参数 expr 的数据类型为 NCHAR 或 NVARCHAR2，则返回 NVARCHAR2 类型。

示例

以下语句比较了字符串的大小，并返回了其中最大的字符串：

```
SELECT GREATEST('HAPPY', 'HAPPEN', 'HAPPINESS')"Greatest"
```

```
FROM DUAL;
```

返回结果：

```
+----------+
| Greatest |
+----------+
| HAPPY |
+----------+
```

以下语句比较了整数 **1** 和 字符串 **3.935**、**2.4** 三者间的大小，由于第一个参数的数据类型为数值数据类型，所以其余参数将被隐式转换为数值数据类型后再进行比较：

```
SELECT GREATEST (1, '3.935', '2.4')"Greatest"
FROM DUAL;
```

返回结果：

```
+----------+
| Greatest |
+----------+
| 3.935 |
+----------+
```

6.6.2 LEAST

LEAST 函数返回一个或多个表达式列表中的最小值。 数据库使用第一个参数 expr 来确定返回类型。 如果其余参数的数据类型和第一个参数 expr 的数据类型不同，则 OceanBase 将第一个参数 expr 之后的每个参数隐式转换为比较之前的第一个 expr 的数据类型。

语法

```
LEAST(expr [, expr ]...)
```

参数

参数	说明
expr	一个表达式或表达式列表。数据类型可以是：NUMBER、FLOAT、BINARY_FLOAT、BINARY_DOUBLE、CHAR、VARCHAR2、NCHAR、NVARCHAR2、或 CLOB。

返回类型

如果第一个参数 expr 的数据类型为 NUMBER、FLOAT、BINARY_FLOAT或 BINARY_DOUBLE，则返回与第一个参数 expr 相同的数据类型。如果第一个参数 expr 的数据类型为 CHAR、VARCHAR2 或 CLOB，则返回 VARCHAR2 类型。如果第一个参数 expr 的数据类型为 NCHAR 或 NVARCHAR2，则返回 NVARCHAR2 类型。

示例

以下语句比较了字符串的大小，并返回了其中最小的字符串：

```
SELECT LEAST('HAPPY', 'HAPPEN', 'HAPPINESS')"Least"
FROM DUAL;
```

返回结果：

```
+--------+
| Least |
+--------+
| HAPPEN |
+--------+
```

以下语句比较了整数 1 和 字符串 3.925、2.4 三者间的大小，由于第一个参数的数据类型为数值数据类型，所以其余参数将被隐式转换为数值数据类型后再进行比较：

```
SELECT LEAST (1, '3.925', '2.4')"Least"
FROM DUAL;
```

返回结果：

```
+-------+
| Least |
+-------+
| 1 |
+-------+
```

6.7 转换函数

6.7.1 CAST

CAST 函数用于将源数据类型的表达式显式转换为另一种数据类型。

语法

```
CAST (expr AS type_name )
```

参数

参数	说明
expr	列名或者表达式。
AS	用于分隔两个参数，在 AS 之前的是要处理的数据，在 AS 之后是要转换的数据类型。
type_name	数据类型可以是 OceanBase 内建数据类型 。

返回类型

返回与 type_name 相同的类型。

下表显示了哪些数据类型可以转换为其他内置数据类型：

	from BINARY_FLOAT, BINARY_DOUBLE	from CHAR, VARCHAR2	from NUMBER	1from DATETIME/INTERVAL	from RAW	from NCHAR, NVARCHAR2
to BINARY_FLOAT, BINARY_DOUBLE	yes	yes	yes	no	no	yes
to CHAR, VARCHAR2	yes	yes	yes	yes	yes	no
to NUMBER	yes	yes	yes	no	no	yes
to DATETIME, INTERVAL	no	yes	no	yes	no	no
to RAW	yes	yes	yes	no	yes	no
to NCHAR, NVARCHAR2	yes	no	yes	yes	yes	yes

1To DATETIME/INTERVAL 数据类型包括 DATE、TIMESTAMP、TIMESTAMP WITH TIMEZONE、INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH。

示例

执行以下语句：

```
SELECT CAST('123' AS INT),CAST(1 AS VARCHAR2(10)),CAST('22-OCT-1997' AS TIMESTAMP WITH LOCAL TIME ZONE)
AS RESULT FROM DUAL;
```

查询结果如下：

```
+-----------------+----------------------+--------------------------+
| CAST('123'ASINT) | CAST(1ASVARCHAR2(10)) | RESULT |
+-----------------+----------------------+--------------------------+
| 123 | 1 | 1997-10-22 00:00:00.000000 |
+-----------------+----------------------+--------------------------+
```

6.7.2 HEXTORAW

HEXTORAW 函数将 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型中包含十六进制数字的字符转换为 RAW 数据类型。

语法

```
HEXTORAW (char)
```

参数

参数	说明
char	十六进制的字符串。字符串类型可为：CHAR、VARCHAR2、NCHAR 或 NVARCHAR2。

返回类型

返回 RAW 类型的数据。

示例

执行以下语句：

```
select HEXTORAW('A123') from dual;
```

查询结果如下：

```
+------------------+
| HEXTORAW('A123') |
+------------------+
| A123 |
+------------------+
```

6.7.3 RAWTOHEX

RAWTOHEX 函数将二进制数转换为一个相应的十六进制表示的字符串。

语法

```
RAWTOHEX (raw)
```

参数

参数	说明
raw	二进制的字符串。

返回类型

十六进制表示的字符串。

示例

执行以下语句：

```
SELECT RAWTOHEX('AB') FROM DUAL;
```

查询结果如下：

```
+----------------+
| RAWTOHEX('AB') |
+----------------+
| 4142 |
+----------------+
```

6.7.4 TO_BINARY_DOUBLE

TO_BINARY_DOUBLE 函数返回一个双精度的 64 位浮点数.

语法

```
TO_BINARY_DOUBLE(expr [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
expr	字符串或 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 数值数据类型。

当 expr 是字符串时，可选的 fmt 和 nlsparam 参数才有效。它们的作用与 TO_CHAR(number) 功能的作用相同。当 expr 为 BINARY_DOUBLE，则该函数返回 expr。

返回类型

双精度的 64 位浮点数。从字符串或 NUMBER 到 BINARY_DOUBLE 的转换可能不准确，因为 NUMBER 和字符类型使用十进制精度表示数值，而 BINARY_DOUBLE 使用二进制精度。从 BINARY_FLOAT 到 BINARY_DOUBLE 的转换是准确的。

示例

执行以下语句：

```
SELECT TO_BINARY_DOUBLE(1222.111) FROM DUAL;
```

查询结果如下：

```
+---------------------------+
| TO_BINARY_DOUBLE(1222.111) |
+---------------------------+
| 1.222111E+003 |
+---------------------------+
```

您可以看 TO_BINARY_FLOAT 和 TO_CHAR(number) 。

6.7.5 TO_BINARY_FLOAT

TO_BINARY_FLOAT 函数返回一个单精度的 32 位浮点数。

语法

```
TO_BINARY_FLOAT(expr [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
expr	字符串或 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 数值数据类型。

当 expr 是字符串时，可选的 fmt 和 nlsparam 参数才有效。它们的作用与 TO_CHAR(number) 功能的作用相同。当 expr 为 BINARY_FLOAT，则函数返回 expr。

返回类型

单精度的 32 位浮点数。从字符串或 NUMBER 到 BINARY_FLOAT 的转换可能不精确，因为 NUMBER 和字符类型使用十进制精度表示数值，而 BINARY_FLOAT 使用二进制精度。如果 BINARY_DOUBLE 值使用的精度比 BINARY_FLOAT 支持的精度高，则从 BINARY_DOUBLE 到 BINARY_FLOAT 的转换是不精确的。

示例

执行以下语句：

```
SELECT TO_BINARY_FLOAT(1222.111) from dual;
```

查询结果如下：

```
+--------------------------+
| TO_BINARY_FLOAT(1222.111) |
+--------------------------+
| 1.22211096E+003 |
+--------------------------+
```

您可以看 TO_BINARY_DOUBLE 和 TO_CHAR(number) 。

6.7.6 TO_CHAR (character)

TO_CHAR (character) 函数将 NCHAR、NVARCHAR2 或 CLOB 数据转换为数据库字符集。

语法

```
TO_CHAR(character )
```

参数

参数	说明
character	可以是 NCHAR、NVARCHAR2 或 CLOB 数据类型。

返回类型

返回 VARCHAR2。当函数将字符 LOB 转换为数据库字符集时，如果要转换的 LOB 值大于目标类型，则数据库返回错误。

示例

新建表 **CLOBTEST**，并给 RAW 数据类型的列插入数据。

```
CREATE TABLE CLOBTEST(TEXT CLOB);
INSERT INTO CLOBTEST VALUES('DWUIDBWUIDBWIOBFWUIOBFIOWBFWUIOBFUWIFB') ;
```

执行以下语句：

```
SELECT TO_CHAR(TEXT) FROM CLOBTEST;
```

查询结果如下：

```
+----------------------------------------+
| TO_CHAR(TEXT) |
+----------------------------------------+
| dwuidbwuidbwiobfwuiobfiowbfwuiobfuwifb |
+----------------------------------------+
```

6.7.7 TO_CHAR (datetime)

TO_CHAR 函数将 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE、TIMESTAMP WITH LOCAL TIME ZONE、INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 等数据类型的值按照参数 fmt 指定的格式转换为 VARCHAR2 数据类型的值。 如果不指定参数 fmt，则参数 datetime 的值将按如下格式转换为 VARCHAR2 数据类型：

- DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE 和 TIMESTAMP WITH LOCAL TIME ZONE 的值被转换为数据库中日期时间值的默认格式。您可在数据类型章节中查看各日期时间类型的默认格式。
- INTERVAL DAY TO SECOND 和 INTERVAL YEAR TO MONTH 数据类型的值转换为数字格式的间隔值。

语法

```
TO_CHAR( datetime [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
datetime	日期时间或间隔数据类型的值。
fmt	输出格式参数，详细格式信息请参见 日期时间格式化的元素 。
nlsparam	用来控制返回的月份和日份所使用的语言。

返回类型

VARCHAR2 数据类型。

示例

示例 1： 以下语句通过 TO_CHAR 函数返回系统当前日期，并且将日期时间值转换为了 **DS DL** 格式：

```
SELECT TO_CHAR(SYSDATE,'DS DL') FROM DUAL;
```

查询结果如下：

```
+----------------------------------+
| TO_CHAR(SYSDATE,'DSDL') |
+----------------------------------+
| 03/08/2020 Sunday, March 08, 2020 |
+----------------------------------+
```

示例 2：以下语句将间隔值转化为指定格式，并且设置了返回语言为 **AMERICAN**：

```
SELECT TO_CHAR(interval'1' year, 'YY-DD', 'nls_language = AMERICAN') FROM DUAL;
```

查询结果如下：

```
+-----------------------------------------------------+
| TO_CHAR(INTERVAL'1'YEAR,'YY-DD','NLS_LANGUAGE=AMERICAN') |
+-----------------------------------------------------+
| +01-00 |
+-----------------------------------------------------+
```

6.7.8 TO_CHAR (number)

TO_CHAR 函数将 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 类型的数值 n 按照指定数值格式 fmt 转换为 varchar2 数据类型的值。如果省略 fmt，则 n 被转换为 VARCHAR2 值，其长度足以保持有效数字。若 n 为负值，则负号显示在输出值最左侧，比如 TO_CHAR(-1,’$9’）返回 -$1，而不是 $-1。

语法

```
TO_CHAR(n [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
n	精确数值或近似数值数据类型类别的表达式。数值数据类型可以为 NUMBER、BINARY_FLOAT 或 BINARY_DOUBLE 数值数据类型。
fmt	输出格式参数，详细格式信息请参见 日期时间格式化的元素 。
nlsparam	支持的语言从 sys.V_$NLS_VALID_VALUES 获取。

fmt 参数列表

fmt 参数	说明
9	返回指定位数的值。
0	它返回前导 0，它返回尾随 0。
,（逗号）	返回指定位置的逗号。您可以在数字格式化中指定多个逗号。 **限制条件**：格式化数值不能以逗号开头，且逗号不能出现在小数字符或句点的右边。
.（小数点）	返回一个小数，且小数点在指定位置。 **限制条件**：在数字格式化中，您只能指定一个小数点。

返回类型

VARCHAR2 数据类型。

示例

执行以下语句：

```
SELECT TO_CHAR(123.456,'999') FROM DUAL;
```

查询结果如下：

```
+------------------------+
| TO_CHAR(123.456,'999') |
+------------------------+
| 123 |
+------------------------+
```

更多关于数据转换的信息，请查看文档 数据转换的安全注意事项 。

6.7.9 TO_DATE

TO_DATE 函数将 CHAR、VARCHAR、NCHAR 或 NVARCHAR2 数据类型的字符转换为日期数据类型的值。

语法

```
TO_DATE(char [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
char	CHAR、VARCHAR、NCHAR 或 NVARCHAR2 字符数据。
fmt	指定参数 char 的日期时间格式化。具体信息请查阅 日期时间格式化 。
nlsparam	由 nls_territory 初始化参数隐式指定或 nls_date_format 参数显式指定。

返回类型

返回DATE类型。

示例

执行以下语句：

```
SELECT TO_DATE('199912','YYYYMM'),TO_DATE('2000.05.20','YYYY.MM.DD'),
(DATE '2008-12-31') XXDATE,
TO_DATE('2008-12-31 12:31:30','YYYY-MM-DD HH24:MI:SS'),
(TIMESTAMP '2008-12-31 12:31:30') XXTIMESTAMP
FROM DUAL;
```

查询结果如下：

```
+---------------------------+-----------------------------------+--------------------+-----------------------------------
-----------------+------------------------------+
| TO_DATE('199912','YYYYMM') | TO_DATE('2000.05.20','YYYY.MM.DD') | XXDATE | TO_DATE('2008-12-
3112:31:30','YYYY-MM-DDHH24:MI:SS') | XXTIMESTAMP |
+---------------------------+-----------------------------------+--------------------+-----------------------------------
-----------------+------------------------------+
| 1999-12-01 00:00:00 | 2000-05-20 00:00:00 | 2008-12-31 00:00:00 | 2008-12-31 12:31:30 | 2008-12-31
12:31:30.000000000 |
+---------------------------+-----------------------------------+--------------------+-----------------------------------
-----------------+------------------------------+
```

6.7.10 TO_DSINTERVAL

TO_DSINTERVAL 函数将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL DAY TO SECOND 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。

语法

```
TO_DSINTERVAL (days hours：minutes：seconds[.frac_secs])
```

参数

参数	说明
days hours：minutes：seconds[.frac_secs]	符合该参数格式的 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串。

返回类型

INTERVAL DAY TO SECOND 数据类型。

示例

以下示例返回了当前时间 100 天后的日期时间值：

```
SELECT SYSDATE+TO_DSINTERVAL('100 00:00:00') FROM DUAL;
```

查询结果如下：

```
+-------------------------------------+
| SYSDATE+TO_DSINTERVAL('10000:00:00') |
+-------------------------------------+
| 2020-06-16 21:26:58 |
+-------------------------------------+
```

6.7.11 TO_NUMBER

TO_NUMBER 函数将 expr 转换为数值数据类型的值。expr 可以是 CHAR、VARCHAR2、NCHAR、NVARCHAR2、BINARY_FLOAT 或 BINARY_DOUBLE 数据类型的数值。

语法

```
TO_NUMBER(expr [, fmt [, 'nlsparam' ] ])
```

参数

参数	说明
expr	CHAR、VARCHAR2、NCHAR、NVARCHAR2、BINARY_FLOAT 或 BINARY_DOUBLE 数据类型的数值。
fmt	格式模型。
nlsparam	支持的语言从 sys.V_$NLS_VALID_VALUES 获取。

返回类型

返回NUMBER类型的数据。

示例

执行以下语句：

```
SELECT TO_NUMBER('199912'),TO_NUMBER('450.05') FROM DUAL;
```

查询结果如下：

```
+--------------------+--------------------+
| TO_NUMBER('199912') | TO_NUMBER('450.05') |
+--------------------+--------------------+
| 199912 | 450.05 |
+--------------------+--------------------+
```

6.7.12 TO_TIMESTAMP

TO_TIMESTAMP 函数将字符串转换为 TIMESTAMP 数据类型。

语法

```
TO_TIMESTAMP (char，[fmt],['nlsparam'])
```

参数

参数	说明
char	字符型。数据类型可以是 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2。
fmt	字符型，指定返回值的格式。详细格式信息请参见 详细格式请参见 日期时间格式化的元素 。
nlsparam	用来控制返回的月份和日份所使用的语言。

返回类型

TIMESTAMP 数据类型。

示例

执行以下语句：

```
SELECT TO_TIMESTAMP ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF')
FROM DUAL;
```

查询结果如下：

```
+---------------------------------------------------------------------+
| TO_TIMESTAMP('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF') |
+---------------------------------------------------------------------+
| 2002-09-10 14:10:10.123000000 |
+---------------------------------------------------------------------+
```

6.7.13 TO_TIMESTAMP_TZ

TO_TIMESTAMP_TZ 函数将字符串转换为 TIMESTAMP WITH TIME ZONE 数据类型，包含时区信息。

语法

```
TO_TIMESTAMP_TZ (char，[fmt],['nlsparam'])
```

参数

参数	说明
char	字符型。数据类型可以是 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2。
fmt	字符型，指定输出格式，详细格式信息请参见 日期时间格式化的元素 。
nlsparam	用来控制返回的月份和日份所使用的语言。

返回类型

TIMESTAMP WITH TIME ZONE 数据类型。

示例

执行以下语句：

```
SELECT TO_TIMESTAMP_TZ ('10-Sep-02 14:10:10.123000', 'DD-Mon-RR HH24:MI:SS.FF') FROM DUAL;
```

查询结果如下：

```
+---------------------------------------------------------------------+
| TO_TIMESTAMP_TZ('10-SEP-0214:10:10.123000','DD-MON-RRHH24:MI:SS.FF') |
+---------------------------------------------------------------------+
| 2002-09-10 14:10:10.123000000 +08:00 |
+---------------------------------------------------------------------+
```

6.7.14 TO_YMINTERVAL

TO_YMINTERVAL 函数将一个 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串转换为一个 INTERVAL YEAR TO MONTH 数据类型的值，该函数可以用来对一个日期时间值进行加减计算。

语法

```
TO_YMINTERVAL (years-months)
```

参数

参数	说明
years-months	符合该参数格式的 CHAR、VARCHAR2、NCHAR 或 NVARCHAR2 数据类型的字符串。

返回类型

INTERVAL YEAR TO MONTH 数据类型。

示例

以下示例展示了返回当前时间 1 年 2 个月后的时间日期值：

```
SELECT SYSDATE,SYSDATE+TO_YMINTERVAL('01-02') FROM DUAL;
```

查询结果如下：

```
+---------------------+-------------------------------+
| SYSDATE | SYSDATE+TO_YMINTERVAL('01-02') |
+---------------------+-------------------------------+
| 2020-03-08 22:32:01 | 2021-05-08 22:32:01 |
+---------------------+-------------------------------+
```

6.8 编码解码函数

6.8.1 DECODE

DECODE 函数依次用参数 search 与 condition 做比较，直至 condition 与 search 的值相等，则返回对应 search 后跟随的参数 result 的值。如果没有 search 与 condition 相等，则返回参数 default 的值。

语法

```
DECODE (condition, search 1, result 1, search 2, result 2 ... search n, result n, default)
```

DECODE 函数的含义可以用 IF...ELSE IF...END 语句进行解释：

```
IF condition = search 1 THEN
RETURN(result 1)
ELSE IF condition = search 2 THEN
RETURN(result 2)
.....
ELSE IF condition = search n THEN
RETURN(result n)
ELSE
RETURN(default)
END IF
```

参数

参数	说明
condition、search 1...search n、result 1...result n、default	数值型（NUMBER、FLOAT、BINARY_FLOAT 或 BINARY_DOUBLE）或字符型（ CHAR、VARCHAR2、NCHAR 或 NVARCHAR2）的表达式。

注意：search 1 ~ search n 不能为条件表达式，这种情况只能用 CASE WHEN THEN END 语句解决：

```
WHEN CASE condition = search 1 THEN
RETURN(result 1)
ELSE CASE condition = search 2 THEN
RETURN(result 2)
.....
ELSE CASE condition = search n THEN
RETURN(result n)
ELSE
RETURN(default)
END
```

示例

示例 1： 使用 DECODE 比较数值的大小。

以下语句使用 DECODE 函数来返回数值 10 与 20 当中较小的那个数。 SIGN() 函数用来计算两个值差的符号，由于 10 小于 20，所以差为负数，SIGN() 返回 -1。此时 DECODE 函数将参数 -1 与 SIGN() 函数的返回值作比较。相等时，返回值 10，不相等时，返回值 20：

```
SELECT DECODE(SIGN(10-20),-1,10,20) FROM DUAL;
```

返回结果如下：

```
+------------------------------+
| DECODE(SIGN(10-20),-1,10,20) |
+------------------------------+
| 10 |
+------------------------------+
```

示例 2：使用 DECODE 函数查看数据中是否包含字符 **S**。

以下语句创建了表 **EMP**，表中包含列 **ename** 和 **sal**，并向其中插入值：

```
CREATE TABLE EMP(ename VARCHAR(30),sal NUMBER);
INSERT INTO EMP VALUES('CLARK', 2750);
INSERT INTO EMP VALUES('KING', 5300);
INSERT INTO EMP VALUES('MILLER', 1600);
INSERT INTO EMP VALUES('ADAMS', 1400);
INSERT INTO EMP VALUES('FORD', 3300);
INSERT INTO EMP VALUES('JONES', 3275);
INSERT INTO EMP VALUES('SCOTT', 3300);
INSERT INTO EMP VALUES('SMITH', 1100);
INSERT INTO EMP VALUES('ALLEN', 1900);
INSERT INTO EMP VALUES('BLAKE', 3150);
INSERT INTO EMP VALUES('JAMES', 1250);
INSERT INTO EMP VALUES('MARTIN', 1550);
INSERT INTO EMP VALUES('TURNER', 1800);
INSERT INTO EMP VALUES('WARD', 1550);
```

以下语句通过函数 INSTR 返回字符 **S** 在列 **ename** 的值中出现的位置，若没有出现过则返回 0。此时 DECODE 函数将 INSTR 函数的返回值与 0 做比较，相等时说明字符 **S** 在值中没有出现过，则 DECODE 函数返回值 **不含有 S**，否则返回值 **含有 S** ：

```
SELECT ENAME, SAL, DECODE(INSTR(ename, 'S'), 0, '不含有 S', '含有 S') AS INFO FROM EMP;
```

查询结果如下：

```
+--------+------+------------+
| ENAME | SAL | INFO |
+--------+------+------------+
| CLARK | 2750 | 不含有 S |
| KING | 5300 | 不含有 S |
| MILLER | 1600 | 不含有 S |
| ADAMS | 1400 | 含有 S |
| FORD | 3300 | 不含有 S |
| JONES | 3275 | 含有 S |
| SCOTT | 3300 | 含有 S |
| SMITH | 1100 | 含有 S |
| ALLEN | 1900 | 不含有 S |
| BLAKE | 3150 | 不含有 S |
| JAMES | 1250 | 含有 S |
| MARTIN | 1550 | 不含有 S |
| TURNER | 1800 | 不含有 S |
| WARD | 1550 | 不含有 S |
+--------+------+------------+
```

6.8.2 ORA_HASH

ORA_HASH 函数获取对应表达式的 HASH 值。

语法

```
ORA_HASH(expr [, max_bucket [, seed_value ] ])
```

参数

参数	说明
expr	通常为数据库表的列名，数据类型可以是数值类型、字符类型、日期时间类型 或 RAW 类型。
max_bucket	可选的 max_bucket 参数确定哈希函数返回的最大桶数。取值范围为 0~4294967295，默认值是 4294967295。
seed_value	可选的 seed_value 参数使 OceanBase 能够为同一组数据产生许多不同的结果。 您可以指定 0~4294967295 之间的任何值。默认值为 0。

返回类型

NUMBER 类型数据。

示例

创建表 **SALE**，并向里面插入数据。执行以下语句：

```
CREATE TABLE SALE(MONTH CHAR(6), SELL NUMBER(10,2));
INSERT INTO SALE VALUES(200001, 1000);
INSERT INTO SALE VALUES(200002, 1100);
INSERT INTO SALE VALUES(200003, 1200);
INSERT INTO SALE VALUES(200004, 1300);
INSERT INTO SALE VALUES(200005, 1400);
INSERT INTO SALE VALUES(200006, 1500);
INSERT INTO SALE VALUES(200007, 1600);
INSERT INTO SALE VALUES(200101, 1100);
INSERT INTO SALE VALUES(200202, 1200);
INSERT INTO SALE VALUES(200301, 1300);
```

使用 ORA_HASH 函数查询 **SALE** 表，并执行以下语句：

```
SELECT ORA_HASH(CONCAT(month,sell),12,0), month, sell FROM Sale;
```

查询结果如下：

```
+-----------------------------------+--------+------+
| ORA_HASH(CONCAT(MONTH,SELL),12,0) | MONTH | SELL |
+-----------------------------------+--------+------+
| 1 | 200001 | 1000 |
| 6 | 200002 | 1100 |
| 5 | 200003 | 1200 |
| 4 | 200004 | 1300 |
| 5 | 200005 | 1400 |
| 2 | 200006 | 1500 |
| 7 | 200007 | 1600 |
| 10 | 200101 | 1100 |
| 7 | 200202 | 1200 |
| 4 | 200301 | 1300 |
+-----------------------------------+--------+------+
```

6.8.3 VSIZE

VSIZE 函数返回 X 的字节大小数。

语法

```
VSIZE(X)
```

数据类型

各种数据类型。

返回类型

返回 x 的字节数，如果 x 为 NULL，则函数返回 NULL。

示例

创建 **employees**，并向里面插入数据。执行以下语句：

```
CREATE TABLE employees(manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(300, 'Wei', '2019-09-11',23600);
INSERT INTO employees VALUES(200, 'Red', '2019-11-05', 23800);
INSERT INTO employees VALUES(100, 'Part', '2018-10-01',24000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',23500);
COMMIT;
```

使用 VSIZE 函数查询 **last_name** 列中 **manager_id = 300** 的字节数。执行以下语句：

```
SELECT last_name, VSIZE (last_name)"BYTES"FROM employees WHERE manager_id = 300;
```

查询结果如下：

```
+-----------+-------+
| LAST_NAME | BYTES |
+-----------+-------+
| Wei | 3 |
+-----------+-------+
```

6.9 空值相关函数

6.9.1 COALESCE

COALESCE 函数返回参数列表中第一个非空表达式，必须指定最少两个参数。

语法

```
COALESCE(expr1, expr2[,..., exprn])
```

参数

参数	说明
expr1, expr2[,..., exprn]	非空表达式，且最少 2 个。

返回类型

返回参数列表中第一个非空表达式，如果所有的参数都是 NULL，则返回 NULL。

示例

假设有一张表 **product_information**，**product_id** 为商品 ID，**list_price** 为该商品原价，**min_price** 为商品最低价，**Sale** 为商品实际售价。设置商品折扣为 9 折，计算各商品的实际售价。此时可使用 COALESCE 函数，若 **list_price** 为空，就按最低价 **min_price** 计算；若 **min_price** 也为空，则按 **5** 计算。

您可以执行以下语句，建立 **product_information** 数据表，并插入数据：

```
CREATE TABLE product_information(supplier_id INT, product_id INT,list_price numeric, min_price numeric);
INSERT INTO PRODUCT_INFORMATION VALUES ('102050', '1659', '45', NULL);
INSERT INTO PRODUCT_INFORMATION VALUES ('102050', '1770', NULL, '70');
INSERT INTO PRODUCT_INFORMATION VALUES ('102050', '2370', '305', '247');
INSERT INTO PRODUCT_INFORMATION VALUES ('102050', '2380', '750', '731');
INSERT INTO PRODUCT_INFORMATION VALUES ('102050', '3255', NULL, NULL);
```

执行以下查询语句：

```
SELECT product_id, list_price,min_price,COALESCE(0.9*list_price, min_price, 5)"Sale"
FROM product_information WHERE supplier_id = 102050 ORDER BY product_id;
```

查询结果如下：

```
+--------------+--------------+------------+--------+
| PRODUCT_ID | LIST_PRICE | MIN_PRICE | Sale |
+--------------+--------------+------------+--------+
| 1659 | 45 | | 40.5 |
+--------------+--------------+------------+--------+
| 1770 | | 70 | 70 |
+--------------+--------------+------------+--------+
| 2370 | 305 | 247 | 274.5 |
+--------------+--------------+------------+--------+
| 2380 | 750 | 731 | 675 |
+--------------+--------------+------------+--------+
| 3255 | | | 5 |
+--------------+--------------+------------+--------+
```

6.9.2 LNNVL

LNNVL 函数判断条件中的一个或者两个操作数是否为 NULL。该函数可以在 WHERE 子句中使用，也可以作为 CASE 表达式中的 WHEN 条件。将条件作为参数，如果条件为 FALSE 或 UNKNOWN，则返回 TRUE；如果条件为 TRUE，则返回 FALSE。

语法

```
LNNVL(condition)
```

参数

参数	说明
condition	条件

假设 a = 2，b 值为 **NULL**，下表显示了 LNNVL 函数的返回值。

条件	条件判断结果	LNNVL 返回值
a = 1	FALSE	TRUE
a = 2	TRUE	FALSE
a IS NULL	FALSE	TRUE
b = 1	UNKNOWN	TRUE
b IS NULL	TRUE	FALSE
a = b	UNKNOWN	TRUE

返回类型

返回布尔型 TRUE 或 FALSE。

示例

假设有一张表 **EMPLOYEES**，给员工姓名列 **name** 和佣金列 **commission_pct** 里面插入数据，执行以下语句：

```
CREATE TABLE EMPLOYEES (name VARCHAR(20), commission_pct numeric);
INSERT INTO EMPLOYEEs VALUES ('Baer', null);
INSERT INTO EMPLOYEEs VALUES ('Bada', null);
INSERT INTO EMPLOYEEs VALUES ('Boll', 0.1);
INSERT INTO EMPLOYEEs VALUES ('Bates', 0.15);
INSERT INTO EMPLOYEEs VALUES ('Eros', null);
INSERT INTO EMPLOYEEs VALUES ('Girl', 0.25);
```

您想知道佣金率低于 20% 的员工人数，包括没有收到佣金的员工。执行以下语句，您只能查询实际获得佣金低于 20% 的员工人数：

```
SELECT COUNT(*) FROM employees WHERE commission_pct < .2;
```

查询结果如下：

```
+------------+
| COUNT(*) |
+------------+
| 2 |
+------------+
```

要包括没有收到佣金的另外 3 名员工，您需要使用 INNVL 函数重写查询。执行以下语句：

```
SELECT COUNT(*) FROM employees WHERE LNNVL(commission_pct >= .2);
```

查询结果如下：

```
+------------+
| COUNT(*) |
+------------+
| 4 |
+------------+
```

6.9.3 NVL

NVL 函数从两个表达式返回一个非 NULL 值。如果 expr1 与 expr2 的结果皆为 NULL 值，则 NVL 函数返回 NULL。

语法

```
NVL(expr1, expr2)
```

参数

参数	说明
expr1	表达式。数据类型可以是 OceanBase 内建数据类型 中的任何数据类型。
expr2	表达式。数据类型可以是 OceanBase 内建数据类型中的任何数据类型。

expr1 和 expr2 必须是相同类型，或者可以隐式转换为相同类型。如果它们不能隐式转换，则 OceanBase 返回错误。隐式转换实现如下：

- 如果 expr1 是 CHAR、NCHAR、NVARCHAR、VARCHAR2 或 VARCHAR 字符型数据，则 OceanBase 在比较 expr1 之前将 expr2 转换为 expr1 的数据类型，并返回 expr1 字符集的 VARCHAR2。
- 如果 expr1 是 NUMBER、FLOAT、BINARY_FLOAT 或 BINARY_DOUBLE 数值型数据，则 OceanBase 确定哪个参数具有最高的数字优先级，隐式地将另一个参数转换为该数据类型，并返回该数据类型。

返回类型

如果 expr1、expr2 是 NULL，则返回 NULL。如果 expr1 是 CHAR、NCHAR、NVARCHAR、VARCHAR2 或 VARCHAR 字符型数据，则返回 expr1 字符集的 VARCHAR2。如果 expr1 是 NUMBER、FLOAT、BINARY_FLOAT 或 BINARY_DOUBLE 数值型数据，则返回 expr1 中具有最高数字优先级的数据类型。

示例

假设有一张表 **EMPLOYEES**，给员工姓名列 **name** 和佣金列 **commission_pct** 里面插入数据，执行以下语句：

```
CREATE TABLE EMPLOYEES (name VARCHAR(20),commission_pct float(3));
INSERT INTO EMPLOYEEs VALUES ('Baer', null);
INSERT INTO EMPLOYEEs VALUES ('Bada', null);
INSERT INTO EMPLOYEEs VALUES ('Boll', 0.1);
INSERT INTO EMPLOYEEs VALUES ('Bates', 0.15);
```

```
INSERT INTO EMPLOYEEs VALUES ('Eric', null);
```

查询员工的姓名和佣金，如果员工没有收到佣金，则显示 **Not Applicable**。执行以下语句：

```
SELECT name, NVL(TO_CHAR(commission_pct), 'Not Applicable') commission
FROM employees WHERE name LIKE 'B%' ORDER BY name;
```

查询结果如下：

```
+--------------+----------------+
| NAME | COMMISSION |
+--------------+----------------+
| Baer | Not Applicable |
+--------------+----------------+
| Bada | Not Applicable |
+--------------+----------------+
| Boll | .1 |
+--------------+----------------+
| Bates | .15 |
+--------------+----------------+
```

6.9.4 NVL2

NVL2 函数根据表达式是否为空，返回不同的值。如果 expr1 不为空，则返回 expr2 的值，如果 expr1 为空，则返回 expr3 的值。expr2 和 expr3 类型不同的话，expr3 会转换为 expr1 的类型。

语法

```
NVL2(expr1, expr2, expr3)
```

参数

参数	说明
expr1	表达式。数据类型可以是 OceanBase 内建数据类型 中的任何数据类型。
expr2	表达式。数据类型可以是 OceanBase 内建数据类型中的任何数据类型。
expr3	表达式。数据类型可以是 OceanBase 内建数据类型中的任何数据类型。

如果 expr2 和 expr3 的数据类型不同，则 OceanBase 将其中一个隐式转换为另一个。 如果它们不能隐式转换，则数据库返回错误。 如果 expr2 是字符或数字数据，则隐式转换规则如下：

- 如果 expr2 是 CHAR、NCHAR、NVARCHAR、VARCHAR2 或 VARCHAR 字符型数据，则 OceanBase 在返回值之前将 expr3 转换为 expr2 的数据类型，除非 expr3 是 NULL。 在这种情况下，不需要数据类型转换，数据库返回 expr2 字符集的 VARCHAR2。
- 如果 expr2 是 NUMBER、FLOAT、BINARY_FLOAT 或 BINARY_DOUBLE 数值型数据，则 OceanBase 确定哪个参数具有最高的数值优先级，隐式地将另一个参数转换为该数据类型，并返回该数据类型。

返回类型

如果 expr1、expr2 是 NULL，则返回 NULL。如果 expr1 是 CHAR、NCHAR、NVARCHAR、VARCHAR2 或 VARCHAR 字符型数据，则返回 expr1 字符集的 VARCHAR2。如果 expr1 是 NUMBER、FLOAT、BINARY_FLOAT 或 BINARY_DOUBLE 数值型数据，则返回 expr1 中具有最高数字优先级的数据类型。

示例

假设有一张表 **EMPLOYEES**，给员工姓名列 **name**、薪资列 **salary** 和佣金列 **commission_pct** 里面插入数据，执行以下语句：

```
CREATE TABLE EMPLOYEES (name VARCHAR(20),commission_pct numeric);
INSERT INTO EMPLOYEEs VALUES ('Baer', 10000, null);
INSERT INTO EMPLOYEEs VALUES ('Bada', 2800, null);
INSERT INTO EMPLOYEEs VALUES ('Boll', 5600, .25);
INSERT INTO EMPLOYEEs VALUES ('Bates', 7300, .39);
INSERT INTO EMPLOYEEs VALUES ('Broll', 4000, null);
```

使用 NVL2 函数查询员工的总收入。如果员工的 **commission_pct** 列不为空，则员工的收入为工资加佣金，否则，收入仅仅为工资。执行以下语句：

```
SELECT name, salary,NVL2(commission_pct, salary + (salary * commission_pct), salary) income
FROM employees WHERE name like 'B%' ORDER BY name;
```

查询结果如下：

```
+--------------+----------------+----------------+
| NAME | SALARY | INCOME |
+--------------+----------------+----------------+
| Bear | 10000 | 10000 |
+--------------+----------------+----------------+
| Bada | 2800 | 2800 |
+--------------+----------------+----------------+
| Boll | 5600 | 7280 |
+--------------+----------------+----------------+
| Bates | 7300 | 10220 |
+--------------+----------------+----------------+
| Broll | 4000 | 4000 |
+--------------+----------------+----------------+
```

6.10 统计函数

6.10.1 窗口函数说明

分析函数（也叫窗口函数）与聚合函数，都是对行集组（一组行的集合）进行聚合计算，不同的是，聚合函数每组只能返回一个值（一行），而窗口函数每组可以返回多个值（多行）。行集组又称为窗口（Window），由 analytic_clause 定义。而窗口大小取决于实际的行数或逻辑间隔（例如时间）。组内每一行都是基于窗口的逻辑计算的结果。

触发一个分析函数需要特殊的关键字 OVER 来指定窗口。一个窗口包含三个组成部分：

- 分区规范，用于将输入行分裂到不同的分区中。这个过程和 GROUP BY 子句的分裂过程相似。
- 排序规范，用于决定输入数据行在窗口函数中执行的顺序。
- 窗口边界，指定计算数据的窗口边界。默认值为 RANGE UNBOUNDED PRECEDING 。这个边界包含当前分区中所有从开始到目前行所有数据。

分析函数是查询中执行的最后一组操作，除了最后的 ORDER BY 子句。在处理窗口函数之前，必须完成所有 JOIN 以及 WHERE、GROUP BY 和 HAVING 子句。因此，窗口函数只能出现在选择列表或 ORDER BY 子句中。

分析函数通常用于计算累积、移动、居中和报告汇总。

语法

analytic_function

```
analytic_function([ arguments ]) OVER (analytic_clause)
```

analytic_clause

```
[ query_partition_clause ] [ order_by_clause [ windowing_clause ] ]
```

query_partition_clause

```
PARTITION BY { expr[, expr ]... | ( expr[, expr ]... ) }
```

order_by_clause

```
ORDER [ SIBLINGS ] BY{ expr | position | c_alias } [ ASC | DESC ] [ NULLS FIRST | NULLS LAST ] [, { expr | position | c_alias } [ ASC | DESC ][ NULLS FIRST | NULLS LAST ]]...
```

windowing_clause

```
{ ROWS | RANGE } { BETWEEN { UNBOUNDED PRECEDING | CURRENT ROW | value_expr { PRECEDING | FOLLOWING } } AND{ UNBOUNDED FOLLOWING | CURRENT ROW | value_expr { PRECEDING | FOLLOWING } } | { UNBOUNDED PRECEDING | CURRENT ROW| value_expr PRECEDING}}
```

在以下各节中将讨论此语法的语义。

analytic_function

分析函数（analytic_function）指定了分析函数的名称。

arguments

参数（arguments），分析函数采用 0~3 个参数。参数可以是任何数字数据类型或可以隐式转换为数字数据类型的任何非数字数据类型。OceanBase 根据 数据类型优先级 确定具有最高数值优先级的参数，然后将其余参

数隐式转换为这个具有最高数值优先级参数的数据类型。返回类型也是具有最高数值优先级参数的数据类型，除非对单个函数另有说明。

analytic_clause

分析子句（analytic_clause），使用 OVER analytic_clause 指示函数在查询结果集上操作。此子句在 FROM、WHERE、GROUP BY 和 HAVING 子句之后计算。您可以在选择列表或按子句顺序中用此子句指定分析函数。若要筛选基于分析函数的查询结果，请将这些函数嵌套在父查询中，然后筛选嵌套子查询的结果。

注意：

- 您不能通过在分析子句中指定分析函数来嵌套分析函数。但是您可以在子查询中指定分析函数，并在其上计算另一个分析函数。
- 您可以使用用户定义的解析函数以及内置的解析函数指定 analytic_clause。

query_partition_clause

分区子句（query_partition_clause），使用 PARTITION BY 子句将查询结果集划分为基于一个或多个 value_expr 的组。如果省略此子句，则函数将查询结果集的所有行视为单个组。

您可以在同一查询中指定多个分析函数，每个函数通过键具有相同或不同的分区。如果您用 query_partition_clause 指定了一个分析函数，并且被查询的对象具有并行属性，那么函数计算也是并行化的。

value_expr 的有效值是常量、列、非分析函数、函数表达式或涉及其中任何一个的表达式。

order_by_clause

使用排序子句 order_by_clause 指定如何在分区内对数据进行排序。 对于所有分析函数，您可以在多个键上的分区中对值进行排序，每个键由 value_expr 定义，由排序序列限定。

在每个函数中，您可以指定多个排序表达式。当使用对值进行排序的函数时，这样做特别有用。

当 order_by_clause 对多行产生相同的值时，函数的行为如下：

- CUME_DIST、DENSE_RANK、NTILE、PERCENT_RANK 和 RANK 为每一行返回相同的结果。
- 即使有一个基于 order_by_clause 的值，ROW_NUMBER 也会为每一行分配一个不同的值。该值基于行处理的顺序，如果 ORDER BY 不能实现总排序，则该顺序可能是不确定的。
- 对于其他分析函数，它的结果取决于窗口规则。如果您指定了一个带有 RANGE 关键字的逻辑窗口，则函数将为每个行返回相同的结果。如果您用 ROWS 关键字指定了物理窗口，则结果是不确定的。

ORDER BY 子句的限制

使用 ORDER BY 子句会受到以下限制：

- 在分析函数中，order_by_clause 必须使用表达式（expr）。SIBLINGS 关键字无效（仅与分层查询中相关）。位置（position）和列别名（c_alias）也无效。否则，该 order_by_clause 与整个查询或子查询的排序命令相同。

  使用 RANGE 关键字的分析函数可以在其函数的 ORDER BY 子句中使用多个排序键。您需要指定以下窗口：

- RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW，简称 RANGE UNBOUNDED PRECEDING。
- RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
- RANGE BETWEEN CURRENT ROW AND CURRENT ROW
- RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING

除了这四个之外，窗口边界只能在分析函数的 ORDER BY 子句中有一个排序键。 此限制不适用于 ROW 关键字指定的窗口边界。

ASC 或 DESC 关键字

指定排序序列，ASC 指升序，DESC 指降序。默认为升序（ASC）。

NULLS FIRST 或 NULLS LAST 关键字

order_by_clause 中的 nulls first 和 nulls last，OceanBase 现在还不支持。

windowing_clause

窗口函数子句（windowing_clause）有一些分析函数可以使用 windowing_clause。相关的关键字如下。

ROWS 或 RANGE 关键字

这些关键字为每一行定义一个用于计算函数结果的窗口，然后将该函数应用于窗口中的所有行。窗口从上到下通过查询结果集或分区移动。窗口也称为 FRAME，OceanBase 同时支持以下窗口语句：

- ROWS 以物理单位（行）指定窗口。
- RANGE 将窗口指定为逻辑偏移量。默认方式是 RANGE UNBOUNDED PRECEDING。您可以在分析函数中使用窗口函数，而要使用 windowing_clause，就必须添加 order_by_clause。

windowing_clause 若由 RANGE 子句定义窗口边界，那么在 order_by_clause 中您只能指定一个表达式。请参阅 ORDER BY 子句的限制 。具有逻辑偏移的分析函数返回的值总是确定性的。但是具有物理偏移量的分析函数返回的值可能会产生不确定的结果。排序表达式返回唯一的排序才能使具有物理偏移量的分析函数返回确定的值，因此，您必须在 order_by_clause 中指定多个列实现唯一的排序。

BETWEEN ... AND 关键字

使用 BETWEEN ... AND 子句指定窗口的起点和终点。第一个表达式（AND 之前）定义起点，第二个表达式（AND 之后）定义终点。 如果省略 BETWEEN 并仅指定一个终点，则 OceanBase 将其视为起点，并且终点默认为当前行。

UNBOUNDED PRECEDING 关键字

UNBOUNDED PRECEDING 指示窗口从分区的第一行开始。这是起点，不是终点。

UNBOUNDED FOLLOWING 关键字

UNBOUNDED FOLLOWING 表示窗口在分区的最后一行结束。这是终点，不是起点。

CURRENT ROW 关键字

作为起点，CURRENT ROW 指定了窗口从当前行或当前值开始（取决于分别指定了 ROW 还是 RANGE）。在这种情况下，端点不能为 value_expr PRECEDING。 作为终点，CURRENT ROW 指定窗口在当前行或值处结束（分别取决于您是否指定了 ROW 或 RANGE）。在这种情况下，起点不能为 value_expr FOLLOWING。

value_expr PRECEDING 或 value_expr FOLLOWING 关键字

- 如果 value_expr FOLLOWING 是起点，则终点必须是 value_expr FOLLOWING。
- 如果 value_expr PRECEDING 是终点，则起点必须是 value_expr PRECEDING。

如果要定义由时间间隔定义的数值格式的逻辑窗口，则可能需要使用转换函数。

如果您指定 ROWS：

- value_expr 是物理偏移量。则它必须是一个常数或表达式，并且必须计算为正数。
- 如果 value_expr 是起点的一部分，则它必须把起点与终点之前的部分当作一行计算。

如果您指定 RANGE：

- value_expr 是逻辑偏移量。它必须是一个常数或表达式，其结果为正数值或间隔字面量。
- 您只能在 order_by_clause 中指定一个表达式。
- 如果 value_expr 为数值，则 ORDER BY expr 必须为数值或 DATE 数据类型。
- 如果 value_expr 为间隔值，则 ORDER BY expr 必须为 DATE 数据类型。如果您完全省略 windowing_clause，则默认值为 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW。

6.10.2 AVG

AVG 函数返回数值列的平均值。

语法

```
AVG([ DISTINCT | ALL ] expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause)。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
DISTINCT	查询时去除数据中的重复值，且忽略数据中的 NULL 值。
ALL	查询时不去除数据中的重复值，且忽略数据中的 NULL 值。ALL 为默认值。
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）或者可以转换成数值类型的表达式。
OVER	使用 OVER 子句定义窗口进行计算。

注意：如果您指定了 DISTINCT 关键字，则 analytic_clause 中不允许出现 order_by_clause 和 windowing_clause 。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT, last_name varchar(50), hiredate varchar(50), SALARY INT);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
COMMIT;
```

执行以下语句计算各列的平均值：

```
SELECT manager_id, last_name, hiredate, salary, AVG(salary) OVER (PARTITION BY manager_id
ORDER BY hiredate ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING) AS c_mavg
FROM employees ORDER BY manager_id, hiredate, salary;
```

返回结果如下：

```
+------------+-----------+------------+--------+---------------------------------------+
| MANAGER_ID | LAST_NAME | HIREDATE | SALARY | C_MAVG |
+------------+-----------+------------+--------+---------------------------------------+
| 100 | Errazuriz | 2017-07-21 | 1400 | 1550 |
| 100 | Raphaely | 2017-07-22 | 1700 | 4700 |
| 100 | De Haan | 2018-05-01 | 11000 | 8900 |
| 100 | Partners | 2018-12-01 | 14000 | 13000 |
| 100 | Hartstein | 2019-05-01 | 14000 | 13833.333 |
| 100 | Weiss | 2019-07-11 | 13500 | 13500 |
| 100 | Russell | 2019-10-05 | 13000 | 13250 |
| 200 | Part | 2018-08-11 | 14000 | 13500 |
| 200 | Bell | 2019-05-25 | 13000 | 13500 |
| 200 | Ross | 2019-06-11 | 13500 | 13250 |
+------------+-----------+------------+--------+---------------------------------------+
```

聚合函数示例

执行以下语句计算 **salary** 的平均值：

```
SELECT AVG(salary) FROM employees;
```

查询结果如下：

```
+----------------------------------------+
| AVG(SALARY) |
+----------------------------------------+
| 10072.727272727272727272727272727272727273 |
+----------------------------------------+
```

6.10.3 COUNT

COUNT 函数用于查询 expr 的行数。

语法

```
COUNT({ * | [ DISTINCT | ALL ] expr }) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
*	表示满足条件的所有行，且包含值为 NULL 的行。
DISTINCT	返回的行中去除重复行，且忽略值为 NULL 的行。
ALL	返回所有值，包含重复行，且忽略值为 NULL 的行。
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）或者可以转换成数值类型的表达式。
OVER	使用 OVER 子句定义窗口进行计算。

注意：

- 对于 COUNT 函数，从不返回 NULL，如果指定了 expr，即返回 expr 不为 NULL 的统计个数，如果指定 COUNT(*) 返回所有行的统计数目。使用参数 DISTINCT 或 ALL 时需要与 expr 用空格隔开。
- 如果您指定了 DISTINCT 关键字，则 analytic_clause 中不允许出现 order_by_clause 和 windowing_clause 。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees(manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(300, 'Wei', '2019-09-11',23600);
INSERT INTO employees VALUES(200, 'Red', '2019-11-05', 23800);
INSERT INTO employees VALUES(100, 'Part', '2018-10-01',24000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',23500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 23000);
INSERT INTO employees VALUES(200, 'Part', '2018-06-11',24500);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
COMMIT;
```

执行以下语句查询表中的行数：

```
SELECT last_name, salary,COUNT(*) OVER (ORDER BY salary RANGE BETWEEN 50 PRECEDING
AND 150 FOLLOWING) AS mov_count FROM employees ORDER BY salary, last_name;
```

查询结果如下：

```
+-----------+--------+-----------+
| LAST_NAME | SALARY | MOV_COUNT |
+-----------+--------+-----------+
| Errazuriz | 1400 | 1 |
| De Haan | 11000 | 1 |
| Hartstein | 14000 | 1 |
| Bell | 23000 | 1 |
| Ross | 23500 | 2 |
| Wei | 23600 | 1 |
| Red | 23800 | 1 |
| Part | 24000 | 1 |
| Part | 24500 | 1 |
+-----------+--------+-----------+
```

聚合函数示例

创建表 **a**，并向里面插入数据。执行以下语句：

```
CREATE TABLE a (
b INT
);
INSERT INTO a VALUES (1);
INSERT INTO a VALUES (null);
INSERT INTO a VALUES (null);
INSERT INTO a VALUES (1);
INSERT INTO a VALUES (null);
INSERT INTO a VALUES (1);
INSERT INTO a VALUES (1);
```

返回表 **a** 值不为 NULL 的统计个数，执行以下语句：

```
SELECT COUNT(b) FROM a;
```

查询结果如下：

```
+----------+
| COUNT(B) |
+----------+
| 4 |
+----------+
```

指定 COUNT(*) 返回所有行的统计数目，执行以下语句：

```
SELECT COUNT(*) FROM a;
```

查询结果如下：

```
+----------+
| COUNT(*) |
+----------+
| 7 |
+----------+
```

6.10.4 SUM

SUM 函数返回参数中指定列的和。此函数将可以隐式转换为数值数据类型的任何数值数据类型或任何非数值数据类型作为参数。 函数返回与参数的数值数据类型相同的数据类型。

语法

```
SUM([ DISTINCT | ALL ] expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
DISTINCT	去除重复行，且忽略值为 NULL 的行。
ALL	返回所有值，包含重复行，且忽略值为 NULL 的行。
expr	可为数值、字符、日期型或其它类型的数据列或表达式。
OVER	使用 OVER 子句定义窗口进行计算。

注意：如果您指定了 DISTINCT 关键字，则 analytic_clause 中不允许出现 order_by_clause 和 windowing_clause 。

返回类型

返回与 expr 相同数据类型的值。

示例

创建表 **employees**，并向里面插入数据，执行以下语句：

```
CREATE TABLE employees(manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(300, 'Wei', '2019-09-11',23600);
INSERT INTO employees VALUES(200, 'Red', '2019-11-05', 23800);
INSERT INTO employees VALUES(100, 'Part', '2018-10-01',24000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',23500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 23000);
INSERT INTO employees VALUES(200, 'Part', '2018-06-11',24500);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
COMMIT;
```

计算工资总额，执行以下语句：

```
SELECT manager_id, last_name, salary, SUM(salary) OVER (PARTITION BY manager_id
ORDER BY salary RANGE UNBOUNDED PRECEDING) l_csum
FROM employees ORDER BY manager_id, last_name, salary, l_csum;
```

查询结果如下：

```
+------------+-----------+--------+--------+
| MANAGER_ID | LAST_NAME | SALARY | L_CSUM |
+------------+-----------+--------+--------+
| 100 | De Haan | 11000 | 12400 |
| 100 | Errazuriz | 1400 | 1400 |
| 100 | Hartstein | 14000 | 26400 |
| 100 | Part | 24000 | 50400 |
| 200 | Bell | 23000 | 23000 |
| 200 | Part | 24500 | 94800 |
| 200 | Red | 23800 | 70300 |
| 200 | Ross | 23500 | 46500 |
| 300 | Wei | 23600 | 23600 |
+------------+-----------+--------+--------+
```

6.10.5 MAX

MAX 函数返回参数中指定的列中的最大值。

语法

```
MAX([ DISTINCT | ALL ] expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
DISTINCT	返回的行中去除重复行，且忽略值为 NULL 的行。
ALL	返回所有值，包含重复行，且忽略值为 NULL 的行。
expr	可为数值、字符、日期型或其它类型的数据列或表达式。
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回与 expr 相同的数据类型值。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT, last_name varchar(50), hiredate varchar(50), SALARY INT);
INSERT INTO employees VALUES(100, 'Wei', '2019-09-11',17000);
INSERT INTO employees VALUES(100, 'Red', '2019-11-05', 17000);
INSERT INTO employees VALUES(101, 'Part', '2018-10-01',12008);
INSERT INTO employees VALUES(102, 'Wei', '2019-09-11',9000);
INSERT INTO employees VALUES(103, 'Red', '2019-11-05', 6000);
INSERT INTO employees VALUES(104, 'Part', '2018-10-01',8000);
COMMIT;
```

执行以下语句查询 **SALARY** 列的最大值：

```
SELECT manager_id, last_name, salary FROM (SELECT manager_id, last_name, salary,
MAX(salary) OVER (PARTITION BY manager_id) AS rmax_sal
FROM employees) WHERE salary = rmax_sal ORDER BY manager_id, last_name, salary;
```

查询结果如下：

```
+------------+-----------+--------+
| MANAGER_ID | LAST_NAME | SALARY |
+------------+-----------+--------+
| 100 | Red | 17000 |
| 100 | Wei | 17000 |
| 101 | Part | 12008 |
| 102 | Wei | 9000 |
| 103 | Red | 6000 |
| 104 | Part | 8000 |
```

```
+------------+-----------+--------+
```

聚合函数示例

执行以下语句查询 **SALARY** 列的最大值：

```
SELECT MAX(salary) FROM employees;
```

查询结果如下：

```
+-------------+
| MAX(SALARY) |
+-------------+
| 17000 |
+-------------+
```

6.10.6 MIN

MIN 函数返回参数中指定列的最小值。

语法

```
MIN([ DISTINCT | ALL ] expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
DISTINCT	返回的行中去除重复行，且忽略值为 NULL 的行。
ALL	返回所有值，包含重复行，且忽略值为 NULL 的行。
expr	可为数值、字符、日期型或其它类型的数据列或表达式。
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回与 expr 相同数据类型的值。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
```

执行以下语句查询 **SALARY** 列的最小值：

```
SELECT manager_id, last_name, hiredate, salary, MIN(salary) OVER(PARTITION BY manager_id
ORDER BY hiredate RANGE UNBOUNDED PRECEDING) AS p_cmin
FROM employees ORDER BY manager_id, last_name, hiredate, salary;
COMMIT;
```

查询结果如下：

```
+------------+-----------+------------+--------+--------+
| MANAGER_ID | LAST_NAME | HIREDATE | SALARY | P_CMIN |
+------------+-----------+------------+--------+--------+
| 100 | De Haan | 2018-05-01 | 11000 | 1400 |
| 100 | Errazuriz | 2017-07-21 | 1400 | 1400 |
| 100 | Hartstein | 2019-05-01 | 14000 | 1400 |
| 100 | Partners | 2018-12-01 | 14000 | 1400 |
| 100 | Raphaely | 2017-07-01 | 1700 | 1700 |
| 100 | Raphaely | 2017-07-22 | 1700 | 1400 |
| 100 | Russell | 2019-10-05 | 13000 | 1400 |
| 100 | Weiss | 2019-07-11 | 13500 | 1400 |
| 200 | Bell | 2019-05-25 | 13000 | 13000 |
| 200 | Part | 2018-08-11 | 14000 | 14000 |
| 200 | Ross | 2019-06-11 | 13500 | 13000 |
+------------+-----------+------------+--------+--------+
```

聚合函数示例

执行以下语句查询 **SALARY** 列的最小值：

```
SELECT MIN(salary) FROM employees ;
```

查询结果如下：

```
+-------------+
| MIN(SALARY) |
+-------------+
| 1400 |
```

```
+-------------+
```

6.10.7 LISTAGG

LISTAGG 函数用于列转行，LISTAGG 对 ORDER BY 子句中指定的每个组内的数据进行排序，然后合并度量列的值。作为单个集合的聚合函数，LISTAGG 对所有行进行操作并返回单个输出行。作为组集聚合，LISTAGG 将对 GROUP BY 子句定义的每个组进行操作并返回输出行。作为分析函数，LISTAGG 基于 query_partition_clause 中的一个或多个表达式将查询结果集分为几组。

语法

```
LISTAGG（measure_expr [,'delimiter']）WITHIN GROUP (order_by_clause)
[OVER query_partition_clause]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。
measure_expr	可以是任何表达式。度量列中的空值将被忽略。
delimiter	指定用于分隔度量值的字符串。此子句是可选的，默认为 NULL。

返回类型

如果度量列是 RAW 的，则返回数据类型为 RAW，否则返回值为 VARCHAR2 型。

示例

分析函数示例

建表 **employees**，并向里面插入数据，执行以下语句：

```
CREATE TABLE employees (department_id INT,manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY
INT);
INSERT INTO employees VALUES(30, 100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(30, 100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(40, 100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(50, 100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(50, 100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(70, 100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(90, 100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(90,100, 'Partners', '2018-12-01',14000);
```

查询 2019 年 10 月 10 日之前雇用的雇员，以及该雇员的部门，雇用日期以及该部门中的其他雇员。执行以下语句：

```
SELECT department_id"Dept", hiredate"Date", last_name"Name",LISTAGG(last_name, '; ') WITHIN GROUP
(ORDER BY hiredate, last_name) OVER (PARTITION BY department_id) as"Emp_list"
FROM employees WHERE hiredate < '2019-10-10' ORDER BY"Dept","Date","Name";
```

查询结果如下：

```
+------+------------+----------+--------------------+
| Dept | Date | Name | Emp_list |
+------+------------+----------+--------------------+
| 30 | 2017-07-01 | Raphaely | Raphaely; De Haan |
| 30 | 2018-05-01 | De Haan | Raphaely; De Haan |
| 40 | 2017-07-21 | Errazuriz | Errazuriz |
| 50 | 2017-07-22 | Raphaely | Raphaely; Hartstein |
| 50 | 2019-05-01 | Hartstein | Raphaely; Hartstein |
| 70 | 2019-07-11 | Weiss | Weiss |
| 90 | 2018-12-01 | Partners | Partners; Russell |
| 90 | 2019-10-05 | Russell | Partners; Russell |
+------+------------+----------+--------------------+
```

聚合函数示例

建表 **employees**，并向里面插入数据，执行以下语句：

```
CREATE TABLE employees (department_id INT,manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY
INT);
INSERT INTO employees VALUES(30, 100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(30, 100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(30, 100, 'Errazuriz', '2017-07-01', 1400);
INSERT INTO employees VALUES(30, 100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(30, 100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(30, 100, 'Weiss', '2019-07-01',13500);
INSERT INTO employees VALUES(30, 100, 'Russell', '2019-07-01', 13000);
INSERT INTO employees VALUES(30,100, 'Partners', '2018-12-01',14000);
```

查询第 30 部门的所有员工，并按雇用日期和姓氏排序。执行以下语句：

```
SELECT LISTAGG(last_name, '; ') WITHIN GROUP (ORDER BY hiredate, last_name) as"Emp_list",
MIN(hiredate) as"Earliest"FROM employees WHERE department_id = 30;
```

查询结果如下：

```
+-----------------------------------------------------------------------------+------------+
| Emp_list | Earliest |
+-----------------------------------------------------------------------------+------------+
| Errazuriz; Raphaely; Raphaely; De Haan; Partners; Hartstein; Russell; Weiss | 2017-07-01 |
+-----------------------------------------------------------------------------+------------+
```

6.10.8 STDDEV

STDDEV 函数用于计算总体标准差。STDDEV 函数将数值型数据作为参数，返回数值型数据。它与函数 STDDEV_SAMP 的不同之处在于，STDDEV 只有一行输入数据时返回 0，而 STDDEV_SAMP 返回 NULL。

OceanBase 中标准差的值是函数 VARIANCE 计算出的方差的算术平方根。

语法

```
STDDEV([ DISTINCT | ALL ] expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
DISTINCT	去重关键字，表示计算唯一值的总体标准差。
ALL	全部数值列
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）或者可以转换成数值类型的表达式。
OVER	使用 OVER 子句定义窗口进行计算。

注意：如果您指定了 DISTINCT 关键字，则 analytic_clause 中不允许出现 order_by_clause 和 windowing_clause 。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees(manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
COMMIT;
```

调用函数并执行以下语句：

```
SELECT last_name, salary, STDDEV(salary) OVER (ORDER BY hiredate)"StdDev"
FROM employees WHERE manager_id = 100 ORDER BY last_name, salary,"StdDev";
```

查询结果如下：

```
+-----------+--------+-----------------------------------------+
| LAST_NAME | SALARY | StdDev |
+-----------+--------+-----------------------------------------+
| De Haan | 11000 | 4702.127178203498995615489088200868644482 |
| Errazuriz | 1400 | 212.132034355964257320253308631454711785 |
| Hartstein | 14000 | 6340.346993658943269176828928801701088079 |
| Partners | 14000 | 6064.899009876421676804205219406952308814 |
| Raphaely | 1700 | 0 |
| Raphaely | 1700 | 173.205080756887729352744634150587236694 |
| Russell | 13000 | 6026.474330580265330900400184969999384459 |
| Weiss | 13500 | 6244.311697171159907069428668980211861012 |
+-----------+--------+-----------------------------------------+
```

聚合函数示例

调用函数并执行以下语句：

```
SELECT STDDEV(salary) FROM employees WHERE manager_id = 100 ;
```

查询结果如下：

```
+-----------------------------------------+
| STDDEV(SALARY) |
+-----------------------------------------+
| 6026.474330580265330900400184969999384459 |
+-----------------------------------------+
```

6.10.9 STDDEV_POP

STDDEV_POP 函数计算总体标准差。STDDEV_POP 函数将数值型数据作为参数，返回数值型数据。

注意： 总体标准差是总体方差的算术平方根。

语法

```
STDDEV_POP(expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）的表达式。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
COMMIT;
```

调用函数并执行以下语句：

```
SELECT manager_id, last_name, salary, STDDEV_POP(salary) OVER (PARTITION BY manager_id) AS pop_std
FROM employees ORDER BY manager_id, last_name, salary, pop_std;
```

查询结果如下：

```
+------------+-----------+--------+-----------------------------------------+
| MANAGER_ID | LAST_NAME | SALARY | POP_STD |
+------------+-----------+--------+-----------------------------------------+
| 100 | De Haan | 11000 | 5637.250548804798333699350384281939588505 |
| 100 | Errazuriz | 1400 | 5637.250548804798333699350384281939588505 |
| 100 | Hartstein | 14000 | 5637.250548804798333699350384281939588505 |
| 100 | Partners | 14000 | 5637.250548804798333699350384281939588505 |
| 100 | Raphaely | 1700 | 5637.250548804798333699350384281939588505 |
| 100 | Raphaely | 1700 | 5637.250548804798333699350384281939588505 |
| 100 | Russell | 13000 | 5637.250548804798333699350384281939588505 |
| 100 | Weiss | 13500 | 5637.250548804798333699350384281939588505 |
| 200 | Bell | 13000 | 408.24829046386301636621401245098189906 9 |
| 200 | Part | 14000 | 408.24829046386301636621401245098189906 9 |
| 200 | Ross | 13500 | 408.24829046386301636621401245098189906 9 |
+------------+-----------+--------+-----------------------------------------+
```

聚合函数示例

调用函数并执行以下语句：

```
SELECT STDDEV_POP(salary) FROM employees ;
```

查询结果如下：

```
+-----------------------------------------+
| STDDEV_POP(SALARY) |
+-----------------------------------------+
| 5249.950806538512715446505486136315088416 |
+-----------------------------------------+
```

6.10.10 STDDEV_SAMP

STDDEV_SAMP 函数计算样本标准差。STDDEV_SAMP 函数将数值型数据作为参数，返回数值型数据。它与函数 STDDEV 的不同之处在于，STDDEV 只有一行输入数据时返回 0，而 STDDEV_SAMP 返回 NULL。

说明： 样本标准差是样本方差的算术平方根。

语法

```
STDDEV_SAMP(expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）或者可以转换成数值类型的表达式。
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
```

```
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
COMMIT;
```

调用函数并执行以下语句：

```
SELECT manager_id, last_name, hiredate, salary,STDDEV_SAMP(salary) OVER (PARTITION BY manager_id
ORDER BY hiredate ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS cum_sdev
FROM employees ORDER BY manager_id, last_name, hiredate, salary, cum_sdev;
```

返回结果如下：

```
+------------+-----------+------------+--------+-----------------------------------------+
| MANAGER_ID | LAST_NAME | HIREDATE | SALARY | CUM_SDEV |
+------------+-----------+------------+--------+-----------------------------------------+
| 100 | De Haan | 2018-05-01 | 11000 | 4702.127178203498995615489088200868644482 |
| 100 | Errazuriz | 2017-07-21 | 1400 | 212.13203435596425732025330863145471178 5 |
| 100 | Hartstein | 2019-05-01 | 14000 | 6340.346993658943269176828928801701088079 |
| 100 | Partners | 2018-12-01 | 14000 | 6064.899009876421676804205219406952308814 |
| 100 | Raphaely | 2017-07-01 | 1700 | NULL |
| 100 | Raphaely | 2017-07-22 | 1700 | 173.2050807568877293527446341505872366 94 |
| 100 | Russell | 2019-10-05 | 13000 | 6026.474330580265330900400184969999384459 |
| 100 | Weiss | 2019-07-11 | 13500 | 6244.311697171159907069428668980211861012 |
| 200 | Bell | 2019-05-25 | 13000 | 707.10678118654752440084436210484903928 5 |
| 200 | Part | 2018-08-11 | 14000 | NULL |
| 200 | Ross | 2019-06-11 | 13500 | 500 |
+------------+-----------+------------+--------+-----------------------------------------+
```

聚合函数示例

调用函数并执行以下语句：

```
SELECT STDDEV_SAMP(salary) FROM employees ;
```

查询结果如下：

```
+-----------------------------------------+
| STDDEV_SAMP(SALARY) |
+-----------------------------------------+
| 5506.194858355615640082358245403620332764 |
+-----------------------------------------+
```

6.10.11 VARIANCE

VARIANCE 函数返回参数指定列的方差。

语法

```
VARIANCE([ DISTINCT | ALL ] expr) [ OVER (analytic_clause) ]
```

作为分析函数使用时，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。作为聚合函数使用时，该函数对一组行的集合进行聚合计算，结果只能返回一个值，此时不需要加 OVER 关键字。

参数

参数	说明
DISTINCT	查询时去除列中的重复值，且忽略列中的 NULL 值。
ALL	查询时不去除列中的重复值，且忽略列中的 NULL 值。ALL 为默认值。
expr	可为数值、字符、日期型或其它类型的数据列或表达式。
OVER	使用 OVER 子句定义窗口进行计算。

注意：如果您指定了 DISTINCT 关键字，则 analytic_clause 中不允许出现 order_by_clause 和 windowing_clause 。

返回类型

返回 NUMBER、 FLOAT、 BINARY_FLOAT 或 BINARY_DOUBLE数值类型。

示例

分析函数示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
COMMIT;
```

执行以下语句计算列 **salary** 的方差：

```
SELECT last_name, salary, VARIANCE(salary) OVER (ORDER BY hiredate)"Variance"
FROM employees WHERE manager_id = 100 ORDER BY last_name, salary,"Variance";
```

查询结果如下：

```
+-----------+--------+----------------------------------------+
| LAST_NAME | SALARY | Variance |
+-----------+--------+----------------------------------------+
| De Haan | 11000 | 22110000 |
| Errazuriz | 1400 | 45000 |
| Hartstein | 14000 | 40200000 |
| Partners | 14000 | 36783000 |
| Raphaely | 1700 | 0 |
| Raphaely | 1700 | 30000 |
| Russell | 13000 | 36318392.857142857142857142857142857142857142857 |
| Weiss | 13500 | 38991428.571428571428571428571428571428571428571 |
+-----------+--------+----------------------------------------+
```

聚合函数示例

执行以下语句计算列 **salary** 的方差：

```
SELECT VARIANCE(salary) FROM employees;
```

查询结果如下：

```
+---------------------------------------+
| VARIANCE(SALARY) |
+---------------------------------------+
| 30318181.818181818181818181818181818182 |
+---------------------------------------+
```

6.10.12 CUME_DIST

CUME_DIST 计算一个值在一组值中的累积分布。返回值的范围为 0 < CUME_DIST <= 1。领带值总是评估到相同的累积分布值。此函数将可以隐式转换为数字数据类型的任何数字数据类型或任何非数字数据类型作为参数。OceanBase 数据库确定具有最高数值优先级的参数，隐式地将其余参数转换为该数据类型，进行计算并返回 NUMBER。

语法

```
CUME_DIST() OVER ([ query_partition_clause ] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它计算指定值在一组值中的相对位置，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）或者可以转换成数值类型的表达式。
{RESPECT | IGNORE} NULLS	表示是否忽略 NULL 值。默认值为 RESPECT NULLS，考虑 NULL 值。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

创建表 **emp_msg**，并向里面插入数据。执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename varchar(30), sal INT, MGR varchar(30));
INSERT INTO emp_msg VALUES(10,'CLARK', 2750, 7839);
INSERT INTO emp_msg VALUES(10,'KING', 5300, NULL);
INSERT INTO emp_msg VALUES(10,'MILLER', 1600, 7782);
INSERT INTO emp_msg VALUES(20,'ADAMS', 1400, 7788);
INSERT INTO emp_msg VALUES(20,'FORD', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'JONES', 3275, 7839);
INSERT INTO emp_msg VALUES(20,'SCOTT', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'SMITH', 1100, 7902);
INSERT INTO emp_msg VALUES(30,'ALLEN', 1900, 7698);
INSERT INTO emp_msg VALUES(30,'BLAKE', 3150, 7839);
INSERT INTO emp_msg VALUES(30,'JAMES', 1250, 7698);
INSERT INTO emp_msg VALUES(30,'MARTIN', 1550, 7698);
INSERT INTO emp_msg VALUES(30,'TURNER', 1800, 7698);
INSERT INTO emp_msg VALUES(30,'WARD', 1550, 7698);
```

返回指定列 **sal** 在同 **deptno** 中的 CUME_DIST 的值，执行以下语句：

```
SELECT deptno , ename , sal, cume_dist ( ) over ( partition BY deptno ORDER BY sal DESC )"RANK"
FROM emp_msg WHERE sal>2000;
```

查询结果如下：

```
+--------+-------+------+-----------------------------------------+
| DEPTNO | ENAME | SAL | RANK |
+--------+-------+------+-----------------------------------------+
| 10 | KING | 5300 | .5 |
| 10 | CLARK | 2750 | 1 |
| 20 | FORD | 3300 | .6666666666666666666666666666666666666667 |
| 20 | SCOTT | 3300 | .6666666666666666666666666666666666666667 |
| 20 | JONES | 3275 | 1 |
| 30 | BLAKE | 3150 | 1 |
+--------+-------+------+-----------------------------------------+
```

6.10.13 DENSE_RANK

DENSE_RANK 计算有序行组中行的秩，并将秩作为 NUMBER 返回。行列是从 1 开始的连续整数，最大的秩值是查询返回的唯一值的数目。在关系的情况下，秩值不被跳过。具有相同值的排序标准的行接收相同的秩。此函数对于顶部 n 和底部 n 报告是有用的。

语法

```
DENSE_RANK( ) OVER([ query_partition_clause ] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回值为 NUMBER、FLOAT、 BINARY_FLOAT 和 BINARY_DOUBLE 等数值类型。

示例

创建表 **emp_msg**，并向里面插入数据。执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename varchar(30), sal INT, MGR varchar(30));
INSERT INTO emp_msg VALUES(10,'CLARK', 2750, 7839);
INSERT INTO emp_msg VALUES(10,'KING', 5300, NULL);
INSERT INTO emp_msg VALUES(10,'MILLER', 1600, 7782);
INSERT INTO emp_msg VALUES(20,'ADAMS', 1400, 7788);
INSERT INTO emp_msg VALUES(20,'FORD', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'JONES', 3275, 7839);
INSERT INTO emp_msg VALUES(20,'SCOTT', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'SMITH', 1100, 7902);
INSERT INTO emp_msg VALUES(30,'ALLEN', 1900, 7698);
INSERT INTO emp_msg VALUES(30,'BLAKE', 3150, 7839);
INSERT INTO emp_msg VALUES(30,'JAMES', 1250, 7698);
INSERT INTO emp_msg VALUES(30,'MARTIN', 1550, 7698);
INSERT INTO emp_msg VALUES(30,'TURNER', 1800, 7698);
INSERT INTO emp_msg VALUES(30,'WARD', 1550, 7698);
```

聚合函数功能示例，执行以下语句：

```
SELECT DENSE_RANK ( 1900 , 7698 ) WITHIN GROUP (ORDER BY sal , mgr) Dense_Rank FROM emp_msg ;
```

查询结果如下：

```
ERROR-00900: You have an error in your SQL syntax;
```

分析函数功能示例，执行以下语句：

```
SELECT deptno, ename, sal, DENSE_RANK ( ) OVER ( partition BY deptno ORDER BY sal DESC )"RANK"FROM emp_msg WHERE sal>2000;
```

查询结果如下：

```
+--------+-------+------+------+
| DEPTNO | ENAME | SAL | RANK |
+--------+-------+------+------+
| 10 | KING | 5300 | 1 |
| 10 | CLARK | 2750 | 2 |
| 20 | SCOTT | 3300 | 1 |
| 20 | FORD | 3300 | 1 |
| 20 | JONES | 3275 | 2 |
| 30 | BLAKE | 3150 | 1 |
+--------+-------+------+------+
```

6.10.14 FIRST_VALUE

FIRST_VALUE 是一个分析函数。它返回有序值中的第一个值。如果集合中的第一个值为 NULL，则函数返回NULL，除非您指定 IGNORE NULLS，该设置对于数据致密化很有用。

语法

```
FIRST_VALUE { (expr) [ {RESPECT | IGNORE} NULLS ] | (expr [ {RESPECT | IGNORE} NULLS ]) } OVER (analytic_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
expr	参数类型不限。
OVER	使用 OVER 子句定义窗口进行计算。
{RESPECT | IGNORE} NULLS	表示是否忽略 NULL 值。默认值为 RESPECT NULLS，考虑 NULL 值。
FROM { FIRST | LAST }	确定计算方向是从窗口的第一行还是最后一行开始，默认值为 FROM FIRST。如果您指定 IGNORE NULLS，则 FIRST_VALUE 返回集合中的第一个非空值，如果所有值都为 NULL，则返回 NULL。

返回类型

数据类型不限。

示例

建表 **emp_msg**，并向里面插入数据。执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename VARCHAR(30), sal INT, MGR VARCHAR(30));
INSERT INTO emp_msg VALUES(10,'CLARK', 2750, 7839);
INSERT INTO emp_msg VALUES(10,'KING', 5300, NULL);
INSERT INTO emp_msg VALUES(10,'MILLER', 1600, 7782);
INSERT INTO emp_msg VALUES(20,'ADAMS', 1400, 7788);
INSERT INTO emp_msg VALUES(20,'FORD', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'JONES', 3275, 7839);
INSERT INTO emp_msg VALUES(20,'SCOTT', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'SMITH', 1100, 7902);
INSERT INTO emp_msg VALUES(30,'ALLEN', 1900, 7698);
INSERT INTO emp_msg VALUES(30,'BLAKE', 3150, 7839);
INSERT INTO emp_msg VALUES(30,'JAMES', 1250, 7698);
INSERT INTO emp_msg VALUES(30,'MARTIN', 1550, 7698);
INSERT INTO emp_msg VALUES(30,'TURNER', 1800, 7698);
INSERT INTO emp_msg VALUES(30,'WARD', 1550, 7698);
```

查询 **emp_msg**表中 **sal** 列最高的第一个非空 **MGR** 值作为 **first_MGR** 列。

```
SELECT deptno , ename , sal , MGR ,
FIRST_VALUE ( MGR ) IGNORE NULLS over ( ORDER BY sal DESC ROWS UNBOUNDED PRECEDING ) AS first_MGR
FROM emp_msg ORDER BY deptno , ename;
```

查询结果如下：

```
+--------+--------+------+------+-----------+
| DEPTNO | ENAME | SAL | MGR | FIRST_MGR |
+--------+--------+------+------+-----------+
| 10 | CLARK | 2750 | 7839 | 7566 |
| 10 | KING | 5300 | NULL | NULL |
| 10 | MILLER | 1600 | 7782 | 7566 |
| 20 | ADAMS | 1400 | 7788 | 7566 |
| 20 | FORD | 3300 | 7566 | 7566 |
| 20 | JONES | 3275 | 7839 | 7566 |
| 20 | SCOTT | 3300 | 7566 | 7566 |
| 20 | SMITH | 1100 | 7902 | 7566 |
| 30 | ALLEN | 1900 | 7698 | 7566 |
| 30 | BLAKE | 3150 | 7839 | 7566 |
| 30 | JAMES | 1250 | 7698 | 7566 |
| 30 | MARTIN | 1550 | 7698 | 7566 |
| 30 | TURNER | 1800 | 7698 | 7566 |
| 30 | WARD | 1550 | 7698 | 7566 |
+--------+--------+------+------+-----------+
```

6.10.15 LAG

LAG 是一个分析函数。它同时提供对多行表的访问，而不需要自连接。给定从查询返回的一系列行和游标的位置，LAG 可以访问位于该位置之前给定物理偏移量的行。您可以给偏移参数指定一个大于零的整数。如果不指定偏移量，则其默认值为 1。如果偏移量超出窗口的范围，则返回可选值。如果不指定默认值，则其默认值为 NULL。

语法

```
LAG { (value_expr [,offset [,default]]) [RESPECT|IGNORE] NULLS
|(value_expr [RESPECT | IGNORE] NULLS [,offset [,default] ]) }
OVER([query_partition_clause] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
value_expr	是要做比对的字段。您不能使用 LAG 函数或其他分析函数来嵌套 value_expr。
offset	是 value_expr 的可选参数偏移量。
default	如果未指定默认值，则其默认值为 NULL。如果在 LEAD 没有显式的设置 default 值的情况下，返回值为 NULL。
{RESPECT | IGNORE} NULLS	表示是否忽略 NULL 值。默认值为 RESPECT NULLS，考虑 NULL 值。
OVER	使用 OVER 子句定义窗口进行计算。

注意： LAG 函数后必须跟 order_by_clause ， query_partition_clause 是可选的。

返回类型

返回值的数据类型不限。

示例

创建 **emp_msg** 表，并向里面插入数据。执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename varchar(30),sal INT);
INSERT INTO emp_msg VALUES(20,'ADAMS',1400);
INSERT INTO emp_msg VALUES(30,'ALLEN',1900);
INSERT INTO emp_msg VALUES(30,'BLAKE',3135);
INSERT INTO emp_msg VALUES(10,'CLARK',2750);
INSERT INTO emp_msg VALUES(20,'FORD',3300);
INSERT INTO emp_msg VALUES(30,'JAMES',1250);
INSERT INTO emp_msg VALUES(20,'JONES',3275);
INSERT INTO emp_msg VALUES(10,'KING',5300);
INSERT INTO emp_msg VALUES(30,'MARTIN',1550);
INSERT INTO emp_msg VALUES(10,'MILLER',1600);
INSERT INTO emp_msg VALUES(20,'SCOTT',3300);
INSERT INTO emp_msg VALUES(20,'SWITH',1100);
INSERT INTO emp_msg VALUES(30,'TURNER',1800);
INSERT INTO emp_msg VALUES(30,'WARD',1550);
```

查询 **emp_msg** 表，最后 5 个值用 **Jane** 代替，从倒数第 6 个值开始追加按 **ename** 字段升序排列的值。执行以下语句：

```
SELECT deptno, ename, sal, LAG(ename,5,'Jane') OVER (ORDER BY ename) AS new_ename
FROM emp_msg;
```

查询结果如下：

```
+--------+--------+------+-----------+
| DEPTNO | ENAME | SAL | NEW_ENAME |
+--------+--------+------+-----------+
| 20 | ADAMS | 1400 | Jane |
| 30 | ALLEN | 1900 | Jane |
| 30 | BLAKE | 3135 | Jane |
| 10 | CLARK | 2750 | Jane |
| 20 | FORD | 3300 | Jane |
| 30 | JAMES | 1250 | ADAMS |
| 20 | JONES | 3275 | ALLEN |
| 10 | KING | 5300 | BLAKE |
| 30 | MARTIN | 1550 | CLARK |
| 10 | MILLER | 1600 | FORD |
| 20 | SCOTT | 3300 | JAMES |
| 20 | SWITH | 1100 | JONES |
| 30 | TURNER | 1800 | KING |
| 30 | WARD | 1550 | MARTIN |
+--------+--------+------+-----------+
```

6.10.16 LAST_VALUE

LAST_VALUE 是一个分析函数。它返回一组有序值中的最后一个值。如果集合中的最后一个值为 NULL，则该函数将返回为 NULL，除非您指定 IGNORE NULLS，此设置对数据致密化很有用。

语法

```
LAST_VALUE {(expr) [RESPECT|IGNORE NULLS] | (expr [RESPECT|IGNORE NULLS])}
OVER (analytic_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。
expr	不能将 LAST_VALUE 或其他分析函数用于 expr 来嵌套分析函数。
FROM { FIRST | LAST }	确定计算方向是从窗口的第一行还是最后一行开始，默认值为 FROM FIRST。
{RESPECT | IGNORE} NULLS	表示是否忽略 NULL 值。默认值为 RESPECT NULLS，考虑 NULL 值。如果您指定 IGNORE NULLS，则 LAST_VALUE 返回集合中的最后一个非空值，如果所有值都为 NULL，则返回 NULL。

返回类型

返回值的数据类型不限。

示例

建表 **emp_msg**，并向里面插入数据，执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename varchar(30),sal INT, MGR varchar(30));
INSERT INTO emp_msg VALUES(10,'CLARK', 2750, 7839);
INSERT INTO emp_msg VALUES(10,'KING', 5300, NULL);
INSERT INTO emp_msg VALUES(10,'MILLER', 1600, 7782);
INSERT INTO emp_msg VALUES(20,'ADAMS', 1400, 7788);
INSERT INTO emp_msg VALUES(20,'FORD', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'JONES', 3275, 7839);
INSERT INTO emp_msg VALUES(20,'SCOTT', 3300, 7566);
INSERT INTO emp_msg VALUES(20,'SMITH', 1100, 7902);
INSERT INTO emp_msg VALUES(30,'ALLEN', 1900, 7698);
INSERT INTO emp_msg VALUES(30,'BLAKE', 3150, 7839);
INSERT INTO emp_msg VALUES(30,'JAMES', 1250, 7698);
INSERT INTO emp_msg VALUES(30,'MARTIN', 1550, 7698);
INSERT INTO emp_msg VALUES(30,'TURNER', 1800, 7698);
INSERT INTO emp_msg VALUES(30,'WARD', 1550, 7698);
```

查询 **emp_msg** 表中 **sal** 列最低的最后一个非空 **MGR** 值作为 **last_MGR** 列，执行以下语句：

```
SELECT deptno , ename , sal , MGR ,
LAST_VALUE ( MGR ) IGNORE NULLS OVER (ORDER BY sal DESC ROWS BETWEEN UNBOUNDED PRECEDING AND
UNBOUNDED FOLLOWING ) AS last_MGR
FROM emp_msg ORDER BY deptno , ename ;
```

查询结果如下：

```
+--------+--------+------+------+-----------+
| DEPTNO | ENAME | SAL | MGR | FIRST_MGR |
+--------+--------+------+------+-----------+
| 10 | CLARK | 2750 | 7839 | 7839 |
| 10 | KING | 5300 | NULL | NULL |
| 10 | MILLER | 1600 | 7782 | 7782 |
| 20 | ADAMS | 1400 | 7788 | 7788 |
| 20 | FORD | 3300 | 7566 | 7566 |
| 20 | JONES | 3275 | 7839 | 7839 |
| 20 | SCOTT | 3300 | 7566 | 7566 |
| 20 | SMITH | 1100 | 7902 | 7902 |
| 30 | ALLEN | 1900 | 7698 | 7698 |
| 30 | BLAKE | 3150 | 7839 | 7839 |
| 30 | JAMES | 1250 | 7698 | 7698 |
| 30 | MARTIN | 1550 | 7698 | 7698 |
| 30 | TURNER | 1800 | 7698 | 7698 |
| 30 | WARD | 1550 | 7698 | 7698 |
+--------+--------+------+------+-----------+
```

6.10.17 LEAD

LEAD 是一种分析函数，它提供了对表多行的访问，而无需进行自我连接。给定从查询返回的一些列行和光标的位置，LEAD 提供超出该位置的物理偏移量的行的访问。

语法

```
LEAD { (value_expr [,offset [,default]]) [RESPECT|IGNORE] NULLS
|(value_expr [RESPECT|IGNORE] NULLS [,offset [,default]]) }
OVER([query_partition_clause] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。
offset	是 value_expr 的可选参数偏移量。
default	如果未指定默认值，则其默认值为 null。如果在 LEAD 没有显式的设置 default 值的情况下，返回值为 NULL。
{RESPECT | IGNORE} NULLS	表示是否忽略 NULL 值。默认值为 RESPECT NULLS，考虑 NULL 值。
value_expr	是要做比对的字段。您不能使用 LEAD 函数或其他分析函数来嵌套 value_expr。

注意： LEAD 函数后必须跟 order_by_clause ， query_partition_clause 是可选的。

返回类型

返回的数据类型不限。

示例

创建 **emp_msg** 表，并向列中插入数据。执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename VARCHAR(30),sal INT);
INSERT INTO emp_msg VALUES(20,'ADAMS',1400);
INSERT INTO emp_msg VALUES(30,'ALLEN',1900);
INSERT INTO emp_msg VALUES(30,'BLAKE',3135);
INSERT INTO emp_msg VALUES(10,'CLARK',2750);
INSERT INTO emp_msg VALUES(20,'FORD',3300);
INSERT INTO emp_msg VALUES(30,'JAMES',1250);
INSERT INTO emp_msg VALUES(20,'JONES',3275);
INSERT INTO emp_msg VALUES(10,'KING',5300);
INSERT INTO emp_msg VALUES(30,'MARTIN',1550);
INSERT INTO emp_msg VALUES(10,'MILLER',1600);
INSERT INTO emp_msg VALUES(20,'SCOTT',3300);
INSERT INTO emp_msg VALUES(20,'SWITH',1100);
INSERT INTO emp_msg VALUES(30,'TURNER',1800);
```

```
INSERT INTO emp_msg VALUES(30,'WARD',1550);
```

查询 **emp_msg** 表，最后 5 个值用 **Jane** 代替，从倒数第 6 个值开始追加按 **ename** 列升序排列的值。

```
SELECT deptno, ename, sal, LEAD(ename,5,'Jane') OVER (ORDER BY ename) AS new_ename
FROM emp_msg;
```

查询结果如下：

```
+--------+--------+------+-----------+
| DEPTNO | ENAME | SAL | NEW_ENAME |
+--------+--------+------+-----------+
| 20 | ADAMS | 1400 | JAMES |
| 30 | ALLEN | 1900 | JONES |
| 30 | BLAKE | 3135 | KING |
| 10 | CLARK | 2750 | MARTIN |
| 20 | FORD | 3300 | MILLER |
| 30 | JAMES | 1250 | SCOTT |
| 20 | JONES | 3275 | SWITH |
| 10 | KING | 5300 | TURNER |
| 30 | MARTIN | 1550 | WARD |
| 10 | MILLER | 1600 | Jane |
| 20 | SCOTT | 3300 | Jane |
| 20 | SWITH | 1100 | Jane |
| 30 | TURNER | 1800 | Jane |
| 30 | WARD | 1550 | Jane |
+--------+--------+------+-----------+
```

6.10.18 NTH_VALUE

NTH_VALUE 返回 analytic_clause 定义的窗口中第 n 行的 measure_expr 值。返回的值具有 measure_expr 的数据类型。

语法

```
NTH_VALUE (measure_expr, n) [ FROM { FIRST | LAST } ][ { RESPECT | IGNORE } NULLS ] OVER (analytic_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。
measure_expr	字段名
n	n 为正数，确定要为其返回测量值的第 n 行，如果 n 是 NULL ，函数将返回错误，如果 n 大于窗口内所有的行数，函数返回 NULL。
FROM { FIRST |	确定计算方向是从窗口的第一行还是最后一行开始，默认值为 FROM FIRST。

LAST }	
{RESPECT | IGNORE} NULLS	表示是否忽略 NULL 值。默认值为 RESPECT NULLS，考虑 NULL 值。

返回类型

返回值的数据类型不限。

示例

创建表 **emp_msg**，并向里面插入数据。执行以下语句：

```
CREATE TABLE emp_msg(deptno INT, ename VARCHAR(30),sal INT, MGR VARCHAR(30), hiredate VARCHAR(50));
INSERT INTO emp_msg VALUES(10,'CLARK', 2750, 7839, '2018-05-01');
INSERT INTO emp_msg VALUES(10,'KING', 5300, NULL, '2018-05-10');
INSERT INTO emp_msg VALUES(10,'MILLER', 1600, 7782, '2018-06-01');
INSERT INTO emp_msg VALUES(20,'ADAMS', 1400, 7788, '2018-05-21');
INSERT INTO emp_msg VALUES(20,'FORD', 3300, 7566, '2018-06-01');
INSERT INTO emp_msg VALUES(20,'JONES', 3275, 7839, '2018-06-20');
INSERT INTO emp_msg VALUES(20,'SCOTT', 3300, 7566, '2018-07-01');
INSERT INTO emp_msg VALUES(20,'SMITH', 1100, 7902, '2018-07-10');
INSERT INTO emp_msg VALUES(30,'ALLEN', 1900, 7698, '2018-08-05');
INSERT INTO emp_msg VALUES(30,'BLAKE', 3150, 7839, '2018-06-10');
INSERT INTO emp_msg VALUES(30,'JAMES', 1250, 7698, '2018-09-05');
INSERT INTO emp_msg VALUES(30,'MARTIN', 1550, 7698, '2018-10-01');
INSERT INTO emp_msg VALUES(30,'TURNER', 1800, 7698, '2019-05-01');
INSERT INTO emp_msg VALUES(30,'WARD', 1550, 7698, '2019-05-10');
```

按部门 **deptno** 分组，查询每个部门的人员的薪资和该部门内排名第三的薪资金额的对比。执行以下语句：

```
SELECT deptno, ename, sal, nth_value(sal, 3) OVER (PARTITION BY deptno ORDER BY sal DESC
rows BETWEEN unbounded preceding AND unbounded following) AS third_most_sal
FROM emp_msg ORDER BY deptno,sal DESC;
```

查询结果如下：

```
+--------+--------+------+---------------+
| DEPTNO | ENAME | SAL | THIRD_MOST_SAL |
+--------+--------+------+---------------+
| 10 | KING | 5300 | 1600 |
| 10 | CLARK | 2750 | 1600 |
| 10 | MILLER | 1600 | 1600 |
| 20 | FORD | 3300 | 3275 |
| 20 | SCOTT | 3300 | 3275 |
| 20 | JONES | 3275 | 3275 |
| 20 | ADAMS | 1400 | 3275 |
| 20 | SMITH | 1100 | 3275 |
| 30 | BLAKE | 3150 | 1800 |
| 30 | ALLEN | 1900 | 1800 |
| 30 | TURNER | 1800 | 1800 |
| 30 | MARTIN | 1550 | 1800 |
```

```
| 30 | WARD | 1550 | 1800 |
| 30 | JAMES | 1250 | 1800 |
+--------+--------+------+----------------+
```

6.10.19 NTILE

NTILE 函数将有序数据集划分为 expr 指示的若干桶，并为每一行分配适当的桶号。桶编号为 1 到 expr。 对于每个分区，expr 值必须解析为正常数。如果 expr 是一个非整数常量，则 OceanBase 将该值截断为整数。 返回值为 NUMBER。

桶中的行数最多可以相差 1，其余值（行数的其余部分除以桶）为每个桶分配一个，从桶 1 开始。如果 expr 大于行数，则将填充与行数相等的多个桶，其余的桶将为空。

您不能通过使用 NTILE 或任何其他分析函数来嵌套分析函数。 但是您可以将其他内置函数表达式用在 expr 中。

语法

```
NTILE(expr) OVER ([query_partition_clause] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
expr	只能为正常数。
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回 NUMBER、 FLOAT、 BINARY_FLOAT 和 BINARY_DOUBLE 数值类型。

示例

根据成绩将同学分成四个等级，为学生评奖。创建表 **course**，并向里面插入数据，执行以下语句：

```
CREATE TABLE course
(
name VARCHAR(8),
grade NUMBER
);
INSERT INTO course VALUES('Linda',50);
INSERT INTO course VALUES('Tan',85);
INSERT INTO course VALUES('Tom',90);
INSERT INTO course VALUES('John',95);
INSERT INTO course VALUES('Mery',55);
INSERT INTO course VALUES('Peter',60);
INSERT INTO course VALUES('Jack',65);
INSERT INTO course VALUES('Rose',70);
```

```
INSERT INTO course VALUES('Tonny',75);
INSERT INTO course VALUES('Apple',80);
COMMIT;
```

执行以下语句：

```
SELECT name, grade, ntile(4) OVER (ORDER BY grade DESC) til FROM course;
```

查询结果如下：

```
+-------+-------+------+
| NAME | GRADE | TIL |
+-------+-------+------+
| John | 95 | 1 |
| Tom | 90 | 1 |
| Tan | 85 | 1 |
| Apple | 80 | 2 |
| Tonny | 75 | 2 |
| Rose | 70 | 2 |
| Jack | 65 | 3 |
| Peter | 60 | 3 |
| Mery | 55 | 4 |
| Linda | 50 | 4 |
+-------+-------+------+
```

6.10.20 PERCENT_RANK

PERCENT_RANK 函数类似于 CUME_DIST（累积分布）函数。 它的返回值范围为 0~1。任何集合中的第一行的 PERCENT_RANK 函数为 0，返回值为 NUMBER。

语法

```
PERCENT_RANK( ) OVER ([query_partition_clause] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

NUMBER 数据类型。

示例

根据成绩将同学分成四个等级，为学生评奖。建表 **course_rank**，并向里面插入数据，执行以下语句：

```
CREATE TABLE course_rank
(
name VARCHAR(8),
id NUMBER
);
INSERT INTO course_rank VALUES('Linda',1);
INSERT INTO course_rank VALUES('Tan',2);
INSERT INTO course_rank VALUES('Tom',3);
INSERT INTO course_rank VALUES('John',4);
INSERT INTO course_rank VALUES('Mery',5);
COMMIT;
```

执行以下语句：

```
SELECT name, id, percent_rank() OVER (ORDER BY id) AS pr1 FROM course_rank;
```

查询结果如下：

```
+-------+------+------+
| NAME | ID | PR1 |
+-------+------+------+
| Linda | 1 | 0 |
| Tan | 2 | .25 |
| Tom | 3 | .5 |
| John | 4 | .75 |
| Mery | 5 | 1 |
+-------+------+------+
```

6.10.21 RANK

RANK 函数基于 OVER 子句中的 ORDER BY 表达式确定一组值的排名。当有相同排序值时，将会有相同的排名，并且值相同的行数会被记录到下个排名中。

语法

```
RANK() OVER ( [ PARTITION BY expr_list ] [ ORDER BY order_list ])
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。
PARTITION BY [col1, col2..]	指定开窗口的列。
ORDER BY col1[asc | desc]	指定排名依据的值。
expr_list	是数值类型或者可以转换成数值类型的类型。

order_list	定义排名值参考的数据列。

示例

建表 **course**，并向 **name** 和 **grade** 列插入数据，执行以下语句：

```
CREATE TABLE course (name VARCHAR(8), grade NUMBER);
INSERT INTO course VALUES('Linda',50);
INSERT INTO course VALUES('Tan',85);
INSERT INTO course VALUES('Tom',90);
INSERT INTO course VALUES('John',95);
INSERT INTO course VALUES('Mery',55);
INSERT INTO course VALUES('Peter',60);
INSERT INTO course VALUES('Jack',65);
INSERT INTO course VALUES('Rose',70);
INSERT INTO course VALUES('Tonny',75);
INSERT INTO course VALUES('Apple',80);
COMMIT;
```

执行以下语句：

```
SELECT name,grade ,RANK() over(ORDER BY grade DESC) FROM course;
```

查询结果如下：

```
+-------+-------+-----------------------------+
| NAME | GRADE | RANK()OVER(ORDERBYGRADEDESC) |
+-------+-------+-----------------------------+
| John | 95 | 1 |
| Tom | 90 | 2 |
| Tan | 85 | 3 |
| Apple | 80 | 4 |
| Tonny | 75 | 5 |
| Rose | 70 | 6 |
| Jack | 65 | 7 |
| Peter | 60 | 8 |
| Mery | 55 | 9 |
| Linda | 50 | 10 |
+-------+-------+-----------------------------+
```

6.10.22 RATIO_TO_REPORT

RATIO_TO_REPORT 函数计算一个值与一组值之和的比率。

语法

```
RATIO_TO_REPORT(expr) OVER ([query_partition_clause])
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请

查阅文档 OVER (analytic_clause) 。

参数

参数	说明
expr	只能为正常数。
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回 NUMBER、 FLOAT、 BINARY_FLOAT 和 BINARY_DOUBLE 数值型数据。

示例

根据部分展示员工产出占部门总展出的比例。建表 **product**，并向里面插入数据，执行以下语句：

```
CREATE TABLE product (name VARCHAR(8), deptno NUMBER, output NUMBER);
INSERT INTO product VALUES('Linda',100,5050);
INSERT INTO product VALUES('Tan',1001,8500);
INSERT INTO product VALUES('Tom',1001,3900);
INSERT INTO product VALUES('John',100,29500);
INSERT INTO product VALUES('Mery',1001,1500);
INSERT INTO product VALUES('Peter',100,1060);
COMMIT;
```

执行以下语句：

```
SELECT name, OUTPUT, deptno, RATIO_TO_REPORT(output) OVER (partition BY deptno) FROM product;
```

查询结果如下：

```
+-------+--------+--------+-----------------------------------------------+
| NAME | OUTPUT | DEPTNO | RATIO_TO_REPORT(OUTPUT)OVER(PARTITIONBYDEPTNO) |
+-------+--------+--------+-----------------------------------------------+
| Linda | 5050 | 100 | .14181409716371805672563886548722269025555 |
| John | 29500 | 100 | .82841898343162033136759337264813254703733 |
| Peter | 1060 | 100 | .02976691940466161190676776186464476270711 |
| Tan | 8500 | 1001 | .61151079136690647482014388489208633095353 |
| Tom | 3900 | 1001 | .28057553956834532374100719424460431654685 |
| Mery | 1500 | 1001 | .10791366906474820143884892086330935251835 |
+-------+--------+--------+-----------------------------------------------+
```

6.10.23 ROW_NUMBER

ROW_NUMBER 函数为应用它的每一行分配一个唯一的数字（无论是分区中的每一行还是查询返回的每一行），按照order_by_clause 中指定的行的有序序列，从 1 开始，通过在检索指定范围的 ROW_NUMBER 的查询中使用子查询嵌套一个子查询，您可以从内部查询的结果中找到一个精确的行子集。 此函数的使用允许您实现 top-n、bottom-n 和 inner-n 报告。对于一致的结果，查询必须确保确定的排序顺序。

语法

```
ROW_NUMBER( ) OVER ([ query_partition_clause ] order_by_clause)
```

作为分析函数使用，您需要使用窗口函数的完整语法，它对一组行的集合进行计算并返回多个值，具体信息请查阅文档 OVER (analytic_clause) 。

参数

参数	说明
OVER	使用 OVER 子句定义窗口进行计算。

返回类型

返回 NUMBER、 FLOAT、 BINARY_FLOAT 和 BINARY_DOUBLE 数值型数据。

示例

以部门为单位根据员工的产出进行排名。建表 **product**，并向里面插入数据。执行以下语句：

```
CREATE TABLE product(name VARCHAR(8), deptno NUMBER, output NUMBER);
INSERT INTO product VALUES('Linda',100,5050);
INSERT INTO product VALUES('Tan',1001,8500);
INSERT INTO product VALUES('Tom',1001,3900);
INSERT INTO product VALUES('John',100,29500);
INSERT INTO product VALUES('Mery',1001,1500);
INSERT INTO product VALUES('Peter',100,1060);
COMMIT;
```

执行以下语句：

```
SELECT name,OUTPUT,deptno,ROW_NUMBER() OVER (partition BY deptno ORDER BY OUTPUT DESC) FROM
product;
```

查询结果如下：

```
+-------+--------+--------+-----------------------------------------------------+
| NAME | OUTPUT | DEPTNO | ROW_NUMBER()OVER(PARTITIONBYDEPTNOORDERBYOUTPUTDESC) |
+-------+--------+--------+-----------------------------------------------------+
| John | 29500 | 100 | 1 |
| Linda | 5050 | 100 | 2 |
| Peter | 1060 | 100 | 3 |
| Tan | 8500 | 1001 | 1 |
| Tom | 3900 | 1001 | 2 |
| Mery | 1500 | 1001 | 3 |
+-------+--------+--------+-----------------------------------------------------+
```

6.10.24 ROLLUP

ROLLUP 函数是聚合函数，它是 GROUP BY 语句的简单扩展。在数据统计和报表生成过程中，它可以为每个分组

返回一个小计，同时为所有分组返回总计，效率比 GROUP BY 和 UNION 组合方法高。

ROLLUP 函数的执行很简单，以下是它的执行顺序：

- 根据参数指定的列从右到左递减分组；
- 对每个分组小计，再对所有分组合计；
- 按照 ORDER BY col1 (，col2，col3，col4 ...) 排序。

如果 ROLLUP 参数为 N 个，则相当于 N+1 个 GROUP BY 分组的 UNION 的结合。

在涉及分组统计的任务中使用 ROLLUP 函数非常高效。例如，沿着时间或地理等层次维度进行小计，您只需要查询 ROLLUP(y, m, day) 或 ROLLUP(country, state, city)。数据仓库管理员使用 ROLLUP 函数可以简化和加快汇总表的维护。

语法

```
SELECT ... GROUP BY ROLLUP(col1 [,col2...])
```

参数

参数	说明
col1	进行分组操作的列名。列数指的是数据库中的行数。

示例

创建表 **group_test**，并向里面插入数据，执行以下语句：

```
CREATE TABLE group_test (group_id int, job varchar2(10), name varchar2(10), salary int);
INSERT INTO group_test VALUES (10, 'Coding', 'Bruce', 1000);
INSERT INTO group_test VALUES (10, 'Programmer', 'Clair', 1000);
INSERT INTO group_test VALUES (20, 'Coding', 'Jason', 2000);
INSERT INTO group_test VALUES (20, 'Programmer', 'Joey', 2000);
INSERT INTO group_test VALUES (30, 'Coding', 'Rebecca', 3000);
INSERT INTO group_test VALUES (30, 'Programmer', 'Rex', 3000);
INSERT INTO group_test VALUES (40, 'Coding', 'Samuel', 4000);
INSERT INTO group_test VALUES (40, 'Programmer', 'Susy', 4000);
COMMIT;
```

对 **group_id** 进行 GROUP BY 分组操作，执行以下语句：

```
SELECT group_id, SUM(salary) FROM group_test GROUP BY group_id;
```

查询结果如下：

```
+----------+-------------+
| GROUP_ID | SUM(SALARY) |
+----------+-------------+
```

```
| 10 | 2000 |
| 20 | 4000 |
| 30 | 6000 |
| 40 | 8000 |
+----------+-------------+
```

对 **group_id** 使用 ROLLUP 函数进行分组，同时求总计。执行以下语句：

```
SELECT group_id, SUM(salary) FROM group_test GROUP BY ROLLUP (group_id);
```

查询结果如下：

```
+----------+-------------+
| GROUP_ID | SUM(SALARY) |
+----------+-------------+
| 10 | 2000 |
| 20 | 4000 |
| 30 | 6000 |
| 40 | 8000 |
| NULL | 20000 |
+----------+-------------+
```

对 **group_id** 列和 **job** 列使用 ROLLUP 函数进行分组，同时求总计。执行以下语句：

```
SELECT group_id, job, SUM(salary) FROM group_test GROUP BY ROLLUP (group_id, job);
```

查询结果如下：

```
+----------+------------+-------------+
| GROUP_ID | JOB | SUM(SALARY) |
+----------+------------+-------------+
| 10 | Coding | 1000 |
| 10 | Programmer | 1000 |
| 10 | NULL | 2000 |
| 20 | Coding | 2000 |
| 20 | Programmer | 2000 |
| 20 | NULL | 4000 |
| 30 | Coding | 3000 |
| 30 | Programmer | 3000 |
| 30 | NULL | 6000 |
| 40 | Coding | 4000 |
| 40 | Programmer | 4000 |
| 40 | NULL | 8000 |
| NULL | NULL | 20000 |
+----------+------------+-------------+
```

上面的 SQL 语句用 GROUP BY 与 UNION 组合方法替换，执行以下语句：

```
SELECT group_id, job, SUM(salary) FROM group_test GROUP BY group_id, job
UNION ALL
```

```
SELECT group_id, NULL, SUM(salary) FROM group_test GROUP BY group_id
UNION ALL
SELECT NULL, NULL, SUM(salary) FROM group_test ORDER BY 1, 2;
```

查询结果如下：

```
+----------+------------+-------------+
| GROUP_ID | JOB | SUM(SALARY) |
+----------+------------+-------------+
| 10 | Coding | 1000 |
| 10 | Programmer | 1000 |
| 10 | NULL | 2000 |
| 20 | Coding | 2000 |
| 20 | Programmer | 2000 |
| 20 | NULL | 4000 |
| 30 | Coding | 3000 |
| 30 | Programmer | 3000 |
| 30 | NULL | 6000 |
| 40 | Coding | 4000 |
| 40 | Programmer | 4000 |
| 40 | NULL | 8000 |
| NULL | NULL | 20000 |
+----------+------------+-------------+
```

输出结果和 ROLLUP 函数一样，但是 ROLLUP 函数更加简洁、高效。

6.10.25 APPROX_COUNT_DISTINCT

APPROX_COUNT_DISTINCT 函数是聚合函数，它对某一列去重后的行数进行计算，结果只能返回一个值，且该值是近似值，该函数可以进一步用于计算被引用的列的选择性。

与函数 COUNT(DISTINCT x) 相比，APPROX_COUNT_DISTINCT 返回的是近似值，所以计算速度极快。在处理大量级的数据时 COUNT(DISTINCT x) 经常要花费很长的时间，使用 APPROX_COUNT_DISTINCT 牺牲了少量的精确度，却换来了计算效率的极大提升。

语法

```
APPROX_COUNT_DISTINCT(expr)
```

参数

参数	说明
expr	是数值类型（NUMBER、FLOAT、BINARY_FLOAT 和 BINARY_DOUBLE）或者可以转换成数值类型的表达式。

返回类型

返回类型与参数 expr 的数据类型相同。

示例

以下语句创建了表 **employees**，并向里面插入数据：

```
CREATE TABLE employees (manager_id INT,last_name varchar(50),hiredate varchar(50),SALARY INT);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-01', 1700);
INSERT INTO employees VALUES(100, 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Errazuriz', '2017-07-21', 1400);
INSERT INTO employees VALUES(100, 'Hartstein', '2019-05-01',14000);
INSERT INTO employees VALUES(100, 'Raphaely', '2017-07-22', 1700);
INSERT INTO employees VALUES(100, 'Weiss', '2019-07-11',13500);
INSERT INTO employees VALUES(100, 'Russell', '2019-10-05', 13000);
INSERT INTO employees VALUES(100, 'Partners', '2018-12-01',14000);
INSERT INTO employees VALUES(200, 'Ross', '2019-06-11',13500);
INSERT INTO employees VALUES(200, 'Bell', '2019-05-25', 13000);
INSERT INTO employees VALUES(200, 'Part', '2018-08-11',14000);
COMMIT;
```

执行以下语句：

```
SELECT last_name, salary, APPROX_COUNT_DISTINCT(salary) OVER (ORDER BY hiredate)"Variance"
FROM employees WHERE manager_id = 100 ORDER BY last_name, salary,"Variance";
```

查询结果如下：

```
+-----------+--------+----------+
| LAST_NAME | SALARY | Variance |
+-----------+--------+----------+
| De Haan | 11000 | 3 |
| Errazuriz | 1400 | 2 |
| Hartstein | 14000 | 4 |
| Partners | 14000 | 4 |
| Raphaely | 1700 | 1 |
| Raphaely | 1700 | 2 |
| Russell | 13000 | 6 |
| Weiss | 13500 | 5 |
+-----------+--------+----------+
```

7 表达式

7.1 SQL 表达式概述

表达式用来计算数据的值，它由一个或多个数值、运算符和 SQL 函数等组件组合而成，通常表达式中要为它的组件假定数据类型。

注意：NLS_COMP 与 NLS_SORT 两个参数值共同影响字符的排序与比较。如果在数据库中 NLS_COMP 的值为 LINGUISTIC，那么本章节中所有被提到的实体都会遵循参数 NLS_SORT 所指定的规则。如果 NLS_COMP 的值没有被指定为 LINGUISTIC，那么函数将不会受 NLS_SORT 的影响。NLS_SORT 的值可以被直接指定，如果没有被指定，它将继承 NLS_LANGUAGE 的值。

下面这个简单表达式的计算结果为 4，数据类型为 NUMBER （与组件的数据类型一致）。

```
2*2
```

如下是一个使用了函数和运算符的复杂表达式。这个表达式表示将当前日期增加 7 天，然后移除时间部分，最后将结果转换成 CHAR 数据类型。

```
TO_CHAR(TRUNC(SYSDATE+7))
```

您还可以在如下情形中使用表达式：

- SELECT 语句的选择列中。
- WHERE 子句中和 HAVING 子句中。
- CONNECT BY、START WITH 和 ORDER BY 子句中。
- INSERT 语句的 VALUES 子句中。
- UPDATE 语句的 SET 子句中。

例如，在下面的 UPDATE 语句的 SET 子句中您可以使用一个表达式替换字符串 **Smith**：

```
SET last_name = 'Smith';
```

这个 SET 子句使用表达式 INITCAP(last_name) 替代字符串 **Smith**：

```
SET last_name = INITCAP(last_name);
```

7.2 简单表达式

一个简单表达式（Simple Expression）可以是列、伪列、常量、序列数或空值。

除了表示用户的数据库对象的集合外，在必须限定表、视图或者物化视图的公共同义词的情况时，schema 也同样可以被指定为 **PUBLIC** 。注意只有在数据操纵语言（DML）语句中支持使用 **PUBLIC** 为公共同义词，而在数据定义语言（DDL）语句中则不支持。

您同样可以只在表中指定 ROWID 而不用在视图或物化视图中指定。NCHAR 与 NVARCHAR2 并不是有效的伪列数据类型。

下列是一些有效的简单表达式：

- employees.last_name
- 'this is a text string'
- 10
- N 'this is an NCHAR string'

7.3 复合表达式

复合表达式（Compound Expression）是由其它类型表达式组合而成的表达式。

您可以使用任意内置函数作为一个表达式。然而，在复合表达式中，一些函数的组合会因为不合适而被弃用。例如，在一个聚合函数中，LENGTH 函数就不适用。

PRIOR 运算符用于层次查询的 CONNECT BY 子句。

下列是一些有效的复合表达式：

- ('CLARK' || 'SMITH')
- LENGTH('MOOSE') * 57
- SORT(144) + 72
- my_fun(TO_CHAR(sysdate, 'DD-MMM-YY'))

7.4 条件表达式

条件表达式（Case Expression）允许用户在不调用存储过程的情况下，在 SQL 语句中使用 IF ... THEN ... ELSE 逻辑。

语法

```
CASE { simple_case_expression
| searched_case_expression
}
[ ELSE else_expr ]
END
```

其中简单条件表达式中 simple_case_expression 的语法为：

```
expr
{ WHEN comparison_expr THEN return_expr }...
```

搜索条件表达式 searched_case_expression 的语法为：

```
{ WHEN condition THEN return_expr }...
```

使用规则

条件的验证

在一个简单条件表达式中，OceanBase 数据库会以 expr 为基准，在 WHEN ... THEN 中搜寻第一个与之等值的 comparison_expr，并返回相应的 return_expr。如果没有 WHEN ... THEN 满足此情形，并且 ELSE 子句存在，那么 OceanBase 数据库会返回 else_expr。否则，OceanBase 会返回 NULL。

在搜索条件表达式中，OceanBase 会自左向右搜索直到 condition 条件成立，然后返回 return_expr。如果所有条件都不成立，且 ELSE 子句存在，则数据库返回 else_expr。否则，数据库会返回 NULL。

条件的计算

OceanBase 数据库使用短路计算法则。对于简单条件表达式，数据库仅会在与 expr 比较之前计算 comparison_expr 的值，而不是在与 expr 比较之前就将计算所有 comparison_expr 的值。因此，如果前一个 comparison_expr 与 expr 相等，OceanBase 将不会计算下一个 comparison_expr 的值。对于搜索条件表达式，数据库将会串行计算每个条件（Condition）是否为真，如果前一个条件（Condition）为真，OceanBase 将不会计算下一个条件。

数据类型

对于简单条件表达式，expr 和所有的 comparison_expr 的值的数据类型必须相同（如 CHAR、VARCHAR2 和 NCHAR； NVARCHAR2、NUMBER 和 BINARY_FLOAT 或 BINARY_DOUBLE），或者都是数值类型。如果所有的返回表达式都是数值类型，则 OceanBase 会选择最高优先级的数据类型，显式地将其它参数转化为此数据类型，并返回这个数据类型。

对于简单条件表达式和搜索条件表达式，所有的 return_exprs 的数据类型必须相同（如 CHAR、VARCHAR2和 NCHAR；NVARCHAR2、NUMBER 和 BINARY_FLOAT；BINARY_DOUBLE），或者都是数值类型。如果所有的返回表达式都是数值类型，则 OceanBase 会选择最高优先级的数据类型，显式地将其它参数转化为此数据类型，并返回这个数据类型。

参数的限制

条件表达式中最多可以拥有 65535 个参数。此限制适用于所有表达式的参数的累计之和，包括简单条件表达式中的初始表达式和可选的 ELSE 表达式。每个 WHEN ... THEN 代表两个参数。为避免超出这个限制，您可以嵌套条件表达式，从而使 return_expr 变成一个条件表达式。

示例

以下示例展示了简单条件表达式的使用：

```
SELECT cust_last_name,
CASE credit_limit
WHEN 100 THEN 'Low'
WHEN 5000 THEN 'High'
ELSE 'Medium' END AS credit
FROM customer
ORDER BY cust_last_name, credit;
```

以下示例展示了搜索条件表达式的使用：

```
SELECT AVG(CASE WHEN e.salary > 2000 THEN e.salary
ELSE 2000 END)"Average Salary"FROM employee e;
```

7.5 列表达式

列表达式（Column Expression）是一种形式受限制的 expr，在本章其他表达式的语法中被命名为 column_expression。列表达式可以是 简单表达式 、 复合表达式 、 函数表达式 或者 表达式列表 ，但它只能包含以下形式的表达式：

- 目标表（被创建，变更或索引的表）的列。

- 常量（字符串或数字）。
- 确定性函数（SQL 内建函数或用户自定义函数）。

除以上形式表达式外，其它形式的表达式都不是有效的列表达式。此外，列表达式不支持使用 PRIOR 关键字的复合表达式与聚合函数。

使用列表达式可以实现以下目的：

- 创建基于函数的索引。
- 显式或隐式定义一个虚拟列。定义一个虚拟列时，column_expression 只适用于在此前语句中已经定义的目标表的列。

列表达式的组件必须是确定的，也就是说，输入同样的值必须返回同样的输出值。

7.6 日期时间表达式

日期时间表达式（Datetime Expression）会生成日期时间数据类型的值。

语法如下：

```
expr AT
{ LOCAL
| TIME ZONE { ' [ + | - ] hh:mi'
| DBTIMEZONE
| 'time_zone_name'
| expr
}
}
```

起始的 expr 可以是除 标量子查询表达式 以外的任意表达式，其计算结果为 TIMESTAMP、TIMESTAMP WITH TIME ZONE 或者 TIMESTAMP WITH LOCAL TIME ZONE 等数据类型的值。DATE 数据类型并不支持。如果这个 expr 自身是日期时间表达式，则它必须在括号中闭合。

日期时间与时间间隔可以组合使用。

如果您指定了 AT LOCAL，OceanBase 将会使用当前会话时区。

AT TIME ZONE 的配置解释如下：

- 字符串 '[+|-] hh:mi' 表示时区偏移量。例如，hh 指定小时数，mi 指定分钟数。
- DBTIMEZONE：OceanBase 使用数据库创建时的数据库时区。
- SESSIONTIMEZONE：OceanBase 使用默认的会话时区或在最新 ALTER SESSION 语句中指定的会话时区。
- time_zone_name：OceanBase 返回 time_zone_name 指定的值作为 datetime_value_expr。
  注意：夏令时特征需要使用时区区域名。这些时区区域名存储在一大一小两种类型的时区文件中。其中一个文件是默认文件，取决于环境以及 OceanBase 的版本。
- expr：如果 expr 返回代表有效时区格式的字符串，则 OceanBase 返回在指定时区中的输入值。否则

，OceanBase 返回一个错误。

以下示例展示了如何将一个时区的日期转换为另一个时区的日期：

```
SELECT FROM_TZ(CAST(TO_DATE('1999-12-01 11:00:00',
'YYYY-MM-DD HH:MI:SS') AS TIMESTAMP), 'America/New_York')
AT TIME ZONE 'America/Los_Angeles'"West Coast Time"
FROM DUAL;
```

7.7 函数表达式

函数表达式（Function Expression）使您可以使用任何内置 SQL 函数或是自定义函数作为表达式。如下所示的是一些有效的内置函数表达式：

- LENGTH('BLAKE')
- ROUND(1234.567*43)
- SYSDATE

注意：自定义函数表达式不能传递对象类型或 XMLType 的参数到远程函数或过程。

自定义函数表达式会调用：

- 一个自定义包、类型或一个独立自定义函数中的函数。
- 一个自定义函数或运算符。

下列是一些有效的自定义函数表达式：

- circle_area(radius)
- payroll.tax_rate(empno)
- hr.employees.comm_pct@remote(dependents, empno)
- DBMS_LOB.getlength(column_name)
- my_function(a_column)

使用自定义函数作为表达式时，支持位置表示法、名字表示法和混合表示法。例如，下面的表示法都是正确的：

```
CALL my_function(arg1 => 3, arg2 => 4) ...
CALL my_function(3, 4) ...
CALL my_function(3, arg2 => 4) ...
```

7.8 间隔表达式

间隔表达式（Interval Expression）生成 INTERVAL YEAR TO MONTH 或 INTERVAL DAY TO SECOND 数据类型的值。

语法如下：

```
( expr1 - expr2 )
{ DAY [ (leading_field_precision) ] TO
SECOND [ (fractional_second_precision) ]
| YEAR [ (leading_field_precision) ] TO
MONTH
}
```

表达式 expr1 和 expr2 可以是计算结果值为 DATE、TIMESTAMP、TIMESTAMP WITH TIME ZONE 或者 TIMESTAMP WITH LOCAL TIME ZONE 数据类型的任意表达式。

leading_field_precision 和 fractional_second_precision 可以是 0 到 9 的任意整数，这两个参数指定了对应元素值的精确度。如果在指定 DAY 和 YEAR 元素时省略了 leading_field_precision 参数，其使用默认值为 2，表示该元素的值不能超过 2 位整数。

如果对 SECOND 元素省略了 fractional_second_ precision 参数，其默认值为 6，表示该值精确到小数点第6位。如果查询的返回值比默认精度位数更多，OceanBase 会返回一个错误。

例如，下列语句用系统时间戳值减去表 **orders** 的 **order_date** 列的值，生成一个间隔值表达式。由于最久远的订单何时生成是未知的，所以指定了 DAY 的精度：

```
SELECT (SYSTIMESTAMP - order_date) DAY(9) TO SECOND FROM orders WHERE order_id = 2458;
```

7.9 对象访问表达式

对象访问表达式（Object Access Expression）指定属性引用与方法调用。

语法如下：

```
{ table_alias.column.
| object_table_alias.
| (expr).
}
{ attribute [.attribute ]...
[.method ([ argument [, argument ]... ]) ]
| method ([ argument [, argument ]... ])
}
```

列参数可以是一个对象或 REF 列。如果指定 expr，则它必须被解析成对象类型。

当一个类型的成员函数在 SQL 语句中被调用，如果 SELF 参数为 NULL，OceanBase 返回 NULL 且函数不会被调用。

以下示例创建了一张基于 order_item_typ 对象类型的表：

```
CREATE TABLE short_orders (
sales_rep VARCHAR2(25), item order_item_typ);
```

```
UPDATE short_orders s SET sales_rep = 'Unassigned';
SELECT o.item.line_item_id, o.item.quantity FROM short_orders o;
```

7.10 标量子查询表达式

标量子查询表达式（Scalar Subquery Expression）是一类从一行返回一列值的子查询。标量子查询表达式的值是子查询的查询列的值。如果子查询返回 0 行，则标量子查询表达式的值是 NULL。如果子查询返回多行，则标量子查询表达式返回一个错误。

在大多数调用表达式的语法中，都可以使用标量子查询表达式。在所有情况下，尽管标量子查询的语法位置已经限定它处在括号内（例如，当标量子查询作为内置函数的参数），标量子查询仍然必须在它所处的括号内闭合。

标量子查询在以下情形中是无效的表达式：

- 作为列的默认值。
- 作为集群的散列表达式。
- 在 DML 语句的 RETURNING 子句中。
- 作为基于函数的索引的基准。
- 在 CHECK 约束中。
- 在 GROUP BY 子句中。
- 在与查询无关的语句中，如 CREATE PROFILE。

7.11 类型构造表达式

类型构造器表达式（Type Constructor Expression）指定了调用构造器方法。类型构造器的参数可以是任意表达式。函数被调用的地方，类型构造器也可以被调用。

语法如下：

```
[ NEW ] [ schema. ]type_name
([ expr [, expr ]... ])
```

NEW 关键字适用于对象类型构造器，但不适用于集合类型构造器。它会引导 OceanBase 通过调用合适的构造器构建一个新的对象。NEW 关键字的使用是可选。

如果 type_name 是对象类型（Object type），则表达式必须是一个有序列表，其中有序列表的第一个参数的值类型与对象类型的第一个属性匹配，第二个参数的值类型与对象类型的第二个属性匹配，以此类推。构造器的参数总数量与对象类型的属性总数量必须匹配。

如果 type_name 是可变长数组（VARRAY）或者嵌套表类型（Nested table type），则表达式列表可以包含 0 个或多个参数。0 参数表示创建一个空集合。否则，每个参数与类型是集合类型中元素的类型的元素值一致。

类型构造器调用的限制

调用类型构造器方法时，尽管对象类型可以拥有超过 999 个属性值，参数的数量不能超过 999。此限制仅适用于从 SQL 中调用构造器。如果是从 PL/SQL 中调用，则需应用 PL/SQL 的限制规则。

示例

以下示例使用了一个预先创建好的类型 **cust_address_typ** 展示了在调用构造器方法时表达式的使用（示例代码 3~11 行为 PL/SQL 语句）：

```
CREATE TYPE address_book_t AS TABLE OF cust_address_typ;

DECLARE
myaddr cust_address_typ := cust_address_typ(
'500 Oracle Parkway', 94065, 'Redwood Shores', 'CA','USA');
alladdr address_book_t := address_book_t();
BEGIN
INSERT INTO customers VALUES (
666999, 'Joe', 'Smith', myaddr, NULL, NULL, NULL, NULL,
NULL, NULL, NULL, NULL, NULL, NULL, NULL);
END;
```

以下示例使用预先创建好的 **warehouse_typ** 类型展示了在调用构造器方法时子查询的使用：

```
CREATE TABLE warehouse_tab OF warehouse_typ;

INSERT INTO warehouse_tab
VALUES (warehouse_typ(101, 'new_wh', 201));

CREATE TYPE facility_typ AS OBJECT (
facility_id NUMBER,
warehouse_ref REF warehouse_typ);

CREATE TABLE buildings (b_id NUMBER, building facility_typ);

INSERT INTO buildings VALUES (10, facility_typ(102,
(SELECT REF(w) FROM warehouse_tab w
WHERE warehouse_name = 'new_wh')));

SELECT b.b_id, b.building.facility_id"FAC_ID",
DEREF(b.building.warehouse_ref)"WH"
FROM buildings b;
```

7.12 表达式列表

表达式列表（Expression List）是一组其它表达式的组合。

表达式列表可以出现在比较和成员条件，以及查询和子查询的 GROUP BY 子句中。在比较和成员条件中的表达式列表有时被称为行值构造器（Row Value constructor）或者行构造器（Row Constructor）。

比较和成员条件出现在 WHERE 子句中。它们可以包含一个或多个逗号分隔的表达式，或是一组或多组表达式，其中每组表达式包含一个或多个逗号分隔的表达式。在接下来的例子中（多组表达式）：

- 每一组被括号绑定。
- 每一组必须包含相同数量的表达式。
- 每一组中表达式的数量需要与比较条件中运算符之前的表达式的数量匹配，或者与成员条件中 IN 关键字之前的表达式的数量匹配。

逗号分隔的表达式列表最多只能包含 1000 个表达式。逗号分隔的表达式组列表可以包含任意数量表达式组，但是每一表达式组最多只能包含 1000 个表达式。

下例是一些有效的表达式列表：

```
(10, 20, 40)
('SCOTT', 'BLAKE', 'TAYLOR')
( ('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA') )
```

在第三个例子中，每一组的表达式数量必须与 SQL 语句条件的第一部分的表达式数量相等。例如：

```
SELECT * FROM employees
WHERE (first_name, last_name, email) IN
(('Guy', 'Himuro', 'GHIMURO'),('Karen', 'Colmenares', 'KCOLMENA'));
```

在简单的 GROUP BY 子句中，使用大写或小写形式的表达式列表都可以：

```
SELECT department_id, MIN(salary) min, MAX(salary) max FROM employees
GROUP BY department_id, salary
ORDER BY department_id, min, max;

SELECT department_id, MIN(salary) min, MAX(salary) max FROM employees
GROUP BY (department_id, salary)
ORDER BY department_id, min, max;
`
```

在 GROUP BY 子句的 ROLLUP、CUBE 和 GROUPING SETS 子句中，您可以在同一表达式列表中将单个表达式与表达式组结合。下例展示了一些在 SQL 语句中有效分组表达式列表：

```
SELECT prod_category, prod_subcategory, country_id, cust_city, count(*)
FROM products, sales, customers
WHERE sales.prod_id = products.prod_id
AND sales.cust_id=customers.cust_id
AND sales.time_id = '01-oct-00'
AND customers.cust_year_of_birth BETWEEN 1960 and 1970
GROUP BY GROUPING SETS (
(prod_category, prod_subcategory, country_id, cust_city), (prod_category, prod_subcategory, country_id),
(prod_category, prod_subcategory),
country_id
)
ORDER BY prod_category, prod_subcategory, country_id, cust_city;
```

8 条件

8.1 SQL 条件概述

条件（Condition）用来判断数据的值，并返回 TRUE、FALSE 或 UNKNOWN。它由一个或多个表达式和逻辑

（布尔）运算符等组件组合而成。只要条件出现在 SQL 语句中，您就必须使用正确的条件语法。

注意： 参数 NLS_COMP 和 NLS_SORT 共同影响字符的排序与比较。若将数据库的 NLS_COMP 指定为LINGUISTIC，那么本开发指南中所有被提到的实体都会遵循参数 NLS_SORT 所指定的规则。若 NLS_COMP 的值没有被指定为 LINGUISTIC，那么函数将不受 NLS_SORT 影响。NLS_SORT 的值可以被直接指定，如果没有被指定，它将继承 NLS_LANGUAGE1 的值。

您可以在这些语句的 WHERE 子句中使用条件：

- DELETE
- SELECT
- UPDATE

您也可以在 SELECT 语句的这些子句中使用条件：

- WHERE
- START WITH
- CONNECT BY
- HAVING

条件可以称为逻辑数据类型，尽管 OceanBase 数据库不正式支持这样的数据类型。

例如，简单条件 1 = 1 判断结果为 TRUE。

下面更复杂的条件将 **salary** 的值加上 **commission_pct** 得值（函数 NVL 将 **salary** 中的 NULL 值替换为 0），并判断总和是否大于 25000。

```
NVL(salary, 0) + NVL(salary + (salary * commission_pct, 0) > 25000)
```

逻辑条件 AND 可以将多个条件组合成一个条件。

```
(1 = 1) AND (5 < 7)
```

SQL 语句中的有效条件

```
name = 'SMITH'
employees.department_id = departments.department_id
hire_date > '01-JAN-08'
job_id IN ('SA_MAN', 'SA_REP')
salary BETWEEN 5000 AND 10000
commission_pct IS NULL AND salary = 2100
```

条件优先

条件优先级指的是 OceanBase 数据库在同一表达式中判断不同条件的顺序。当计算包含多个条件的表达式时，先判断较高优先级的条件，最后判断较低优先级的条件，而优先级相等的条件则按照从左到右的顺序判断。但是 AND 和 OR 连接的多个条件，不能按照优先级进行判断。

SQL 条件优先级表

条件类型	功能
=、!=、<、>、<=、>=	比较。
IS [NOT] NULL、LIKE、[NOT] BETWEEN、[NOT] IN、EXISTS、IS OF 类型	比较。
NOT	取幂，逻辑取反。
AND	比较。
OR	分离。

优先级按照从高到低排列出，同一行上列出的条件具有相同的优先级。

8.2 比较条件

比较条件（Comparison Condition）用于比较一个表达式与另一个表达式，比较的结果为 TRUE、FALSE 或 UNKNOWN。

比较条件不能比较大对象（LOB）数据类型。但是您可以通过 PL/SQL 程序对 CLOB 数据进行比较。

比较数值表达式时，OceanBase 使用数值优先级来确定条件中的数值比较顺序，比如是比较 NUMBER、BINARY_FLOAT 还是 BINARY_DOUBLE 值。

比较字符表达式时，OceanBase 使用 字符数据类型比较规则 中的规定。它规定了表达式的字符集在比较之前是如何统一的，使用 二进制和语言比较 ，以及使用 空白填充和非空白填充的比较语义 。

当使用比较条件对字符值进行语言比较时，首先将它们转换为排序键，然后做比较，比较的过程类似 RAW 数据类型。排序键是由函数 NLSSORT 返回的值。如果两个表达式生成的排序键的前缀相同，即使它们在值的其余部分不同，它们也可以在语言上相等。

如果两个非标量类型的对象具有相同的命名类型，并且它们的元素之间存在一一对应的关系，则它们是可比较的。用户定义的对象类型—嵌套表，在相等或 IN 条件下使用时，必须定义 MAP 方法。嵌套表的元素可以比较。

简单比较条件

一个简单比较条件可以是单个表达式与表达式列表比较或单个表达式与子查询结果比较。

简单比较条件语法如下：

```
expr {= | != | ^= | | < | >= |} ( expression_list | subquery )
```

在简单比较条件中，如果是单个表达式与表达式列表比较，并且表达式列表中的表达式必须在数量和数据类型上与运算符左边的表达式匹配。如果是单个表达式与子查询的值比较，则子查询返回的值必须与运算符左侧的表达式在数量和数据类型上匹配。

组比较条件

一个组比较条件可以是单个表达式与表达式列表或子查询结果的任何或所有成员比较，也可以是多个表达式与表达式列表或子查询结果的任何或所有成员比较。

在组比较条件中，如果是单个表达式或多个表达式列表与表达式列表的任何或所有成员比较，则每个表达式列表中的表达式必须在数量和数据类型上与运算符左边的表达式匹配。如果是单个表达式或多个表达式与子查询结果的任何或所有成员比较，则子查询返回的值必须与运算符左侧的表达式在数量和数据类型上匹配。

组比较条件语法有如下两种：

```
expr {= | != | ^= | <> | < | >= | ANY | SOME | ALL } ({ expression_list | subquery})

(expr [, expr ]...){= | != | ^= | ANY | SOME | ALL} ({expression_list [, expression_list ]... |subquery})
```

8.3 浮点条件

浮点条件（Floating Point Condition）可以帮助您排除表达式不支持的数据类型。其中表达式必须解析为数值数据类型或任何可以隐式转换为数值数据类型的数据类型。

语法

```
expr IS [ NOT ] { NAN | INFINITE }
```

示例

条件类型	功能	示例
IS [NOT] NAN	OceanBase 不支持 NAN，expr 必须加 NOT。	SELECT COUNT(*) FROM employees WHERE commission_pct IS NOT NAN;
IS [NOT] INFINITE	OceanBase 不支持 INFINITF，expr 必须加 NOT。	SELECT last_name FROM employees WHERE salary IS NOT INFINITE;

更多信息

- 浮点数字
- 数据类型转换

8.4 逻辑条件

逻辑条件（Logical Condition）将两个条件组合在一起，产生单个结果或反转单个条件的结果。

逻辑条件 NO

逻辑条件 NO 表示“非”，可以反转单个条件的结果。如果条件为 FALSE，则返回 TRUE。如果条件为 TRUE，则返回 FALSE。如果它是 UNKNOWN，则返回 UNKNOWN。

逻辑条件 NO 示例

```
SELECT * FROM employees WHERE NOT (job_id IS NULL) ORDER BY employee_id;
```

```
SELECT * FROM employees WHERE NOT (salary BETWEEN 1000 AND 2000) ORDER BY employee_id;
```

逻辑条件 AND

逻辑条件 AND 表示“与”，用于连接两个条件。如果两个条件均为 TRUE，则返回 TRUE。如果任意一个为 FALSE，则返回 FALSE。否则返回 UNKNOWN。

逻辑条件 AND 示例

```
SELECT * FROM employees WHERE job_id = 'PU_CLERK' AND department_id = 30 ORDER BY employee_id;
```

逻辑条件 OR

逻辑条件 OR 表示“或”，表示任何一个都可以。如果任一条件为 TRUE，则返回 TRUE。如果两者均为 FALSE，则返回 FALSE。否则返回 UNKNOWN。

逻辑条件 OR 示例

```
SELECT * FROM employees WHERE job_id = 'PU_CLERK' OR department_id = 10 ORDER BY employee_id;
```

8.5 模式匹配条件

模式匹配条件（Pattern-matching Condition）用来比较字符数据。

LIKE 条件

LIKE 条件用于模式匹配。相等运算符（=）指的是一个字符值与另一个字符值完全匹配，而 LIKE 条件通过在第一个值中搜索由第二个字符值指定的模式来将一个字符值的一部分与另一个字符值进行匹配。LIKE 使用输入字符集定义的字符来计算字符串。

语法

```
char1 [NOT] LIKE char2 [ ESCAPE esc_char ]
```

除了 LIKE，特殊的模式匹配字符 _ 表示值中的一个字符恰好匹配，% 表示值中的零个或多个字符匹配。模式 % 不能与 NULL 匹配。

参数

参数	说明
char1	字符表达式，例如字符列，称为搜索值。
char2	字符表达式，通常是文字，称为模式。
esc_char	字符表达式，通常是文字，称为转义字符。当转义符置于模式匹配符之前时，该模式匹配符被解释为普通的字符。

示例

以下语句使用了 LIKE：

```
SELECT last_name FROM employees WHERE last_name LIKE '%A\_B%' ESCAPE '\'
ORDER BY last_name;
```

ESCAPE '\'会将 %A_B% 中 \ 后面的模式匹配符 '_' 解释为普通的字符。

```
SELECT salary FROM employees WHERE 'SM%' LIKE last_name ORDER BY salary;
```

REGEXP_LIKE 条件

REGEXP_LIKE 用于正则表达式匹配。REGEXP_LIKE使用输入字符集定义的字符评估字符串。

语法

```
REGEXP_LIKE(source_char, pattern [, match_param ])
```

参数

参数	说明
source_char	用作搜索值的字符表达式，数据类型可以是 CHAR，VARCHAR2，NCHAR，NVARCHAR2 或 CLOB。
pattern	正则表达式，数据类型可以是 CHAR，VARCHAR2，NCHAR，NVARCHAR2 或 CLOB。
source_char	数据类型 VARCHAR2 或 CHAR 的字符表达式，允许您更改条件的默认匹配行为。

如果模式的数据类型与 source_char 的数据类型不同，OceanBase 将模式转换为 source_char 的数据类型。

示例

创建表 **employees**，并向里面插入数据。执行以下语句：

```
CREATE TABLE employees(manager_id INT, first_name varchar(50), last_name varchar(50), hiredate
varchar(50),SALARY INT);
INSERT INTO employees VALUES(300, 'Steven', 'King', '2019-09-11',23600);
INSERT INTO employees VALUES(200, 'Steven', 'Markle', '2019-11-05', 23800);
INSERT INTO employees VALUES(100, 'Deven', 'Part', '2018-10-01',24000);
INSERT INTO employees VALUES(200, 'Carlos', 'Ross', '2019-06-11',23500);
INSERT INTO employees VALUES(200, 'Teven', 'Bell', '2019-05-25', 23000);
INSERT INTO employees VALUES(200, 'Stephen', 'Stiles', '2018-06-11',24500);
INSERT INTO employees VALUES(100, 'Ame', 'De Haan', '2018-05-01',11000);
INSERT INTO employees VALUES(100, 'Jon', 'Errazuriz', '2017-07-21', 1400);
COMMIT;
```

查询返回名字为 **Steven** 或 **Stephen** 的员工的名字和姓氏（返回列 **first_name** 中以 **Ste** 开头、以 **en** 结尾、中间是 **v** 或 **ph** 的值），执行以下语句：

```
SELECT first_name, last_name FROM employees WHERE REGEXP_LIKE (first_name, '^Ste(v|ph)en$')
ORDER BY first_name, last_name;
```

查询结果如下：

```
+--------------+-------------+
| FIRST_NAME | LAST_NAME |
+--------------+-------------+
| Steven | King |
+--------------+-------------+
| Steven | Markle |
+--------------+-------------+
| Stephen | Stiles |
+--------------+-------------+
```

8.6 空条件

空条件（Null Condition）检测 NULL 值。这是用于检测 NULL 的唯一条件。

语法

```
expr IS [ NOT ] NULL
```

示例

执行以下语句：

```
SELECT last_name FROM employees WHERE commission_pct IS NULL
ORDER BY last_name;
```

8.7 复合条件

复合条件（Compound Condition）指的是其他条件的组合。

语法

```
{ (condition) | NOT condition | condition { AND | OR } condition }
```

更多关于条件 NOT、AND 和 OR 的信息，请参考文档 逻辑条件 。

8.8 BETWEEN 条件

BETWEEN 条件（Between Condition）确定一个表达式的值是否在其他两个表达式定义的间隔内。

语法

```
expr1 [ NOT ] BETWEEN expr2 AND expr3
```

```
NOT (expr1 BETWEEN expr2 AND expr3)
```

expr1、expr2 和 expr3 这三个表达式都必须是数字、字符或日期时间表达式。在 SQL 中，可能会对 expr1 进行多次计算。如果在 PL/SQL 中出现表达式，则保证只计算一次 expr1。如果表达式不都是相同的数据类型，那么 OceanBase 数据库将表达式 隐式转换 为统一的数据类型。否则返回错误。

示例

查询薪资在 2000~3000 范围内的员工信息，并按照员工号排序。

```
SELECT * FROM employees WHERE salary BETWEEN 2000 AND 3000 ORDER BY employee_id;
```

8.9 存在条件

存在条件（Exists Condition）用于测试子查询中是否存在指定的行。

语法

```
EXISTS (subquery)
```

如果子查询至少返回一行，则说明存在您想要的数据。

示例

```
SELECT department_id FROM departments d WHERE EXISTS (SELECT * FROM employees e
WHERE d.department_id = e.department_id) ORDER BY department_id;
```

8.10 IN 条件

IN 条件（In Condition）是成员资格条件。它测试值或子查询列表中成员的值。

语法

```
expr [ NOT ] IN ({ expression_list | subquery })
|
( expr [, expr ]... ) [ NOT ] IN ( { expression_list [, expression_list ]... | subquery } )
```

IN 条件可以测试表达式是表达式列表或子查询的成员，或者多个表达式是表达式列表或子查询的成员。并且每个表达式列表中的表达式必须在数量和数据类型上与运算符 IN 左边的表达式匹配。

示例

IN 示例：相当于 ANY，表示集合中所有的成员。

```
SELECT * FROM employees WHERE job_id IN ('PU_CLERK','SH_CLERK') ORDER BY employee_id;

SELECT * FROM employees WHERE salary IN (SELECT salary FROM employees
WHERE department_id =30) ORDER BY employee_id;
```

NOT IN 示例：相当于 != ANY。如果集合中的任何成员为 NULL，则计算为 false。

```
SELECT * FROM employees WHERE salary NOT IN (SELECT salary FROM employees
WHERE department_id = 30) ORDER BY employee_id;

SELECT * FROM employees WHERE job_id NOT IN ('PU_CLERK', 'SH_CLERK')
ORDER BY employee_id;
```

9 查询和子查询

9.1 查询和子查询概述

查询是数据库用来获取数据的方式，它可搭配条件限制的子句（如 WHERE），排列顺序的子句（如 ORDER BY）等语句来获取查询结果。子查询是指嵌套在一个上层查询中的查询。上层的查询一般被称为父查询或外部查询。子查询的结果作为输入传递回父查询或外部查询。父查询将这个值结合到计算中，以便确定最后的输出。SQL 语言允许多层嵌套查询，即一个子查询中还可以嵌套其他子查询。同时，子查询可以出现在 SQL 语句中的各种子句中，比如 SELECT 语句、FROM 语句和 WHERE 语句等。下列为 SQL 语句中常见的查询：

- 简单查询
- 层次查询
- 集合
- 连接
- 子查询

简单查询（Simple Queries）

简单查询指从 OceanBase 一个或多个选择列表或视图中检索一个或多个列数据的操作，列的数量以及它们的数据类型和长度由选择列表的元素确定。而选择列表指的是 SELECT 关键字之后和 FROM 子句之前的表达式列表。

层次查询（Hierarchical Query）

层次查询是一种具有特殊功能的查询语句，通过它能够将分层数据按照层次关系展示出来。分层数据是指关系表中的数据之间具有层次关系。

集合

您可以使用集合运算符 UNION、UNION ALL、INTERSECT 和 MINUS 来组合多个查询。 所有集合运算符都具有相同的优先级。如果 SQL 语句包含多个集合运算符，则 OceanBase 从左到右对它们进行判断，除非括号中指定

了顺序。本节主要讲了一家集合运算符：

运算符	说明
UNION	返回两个结果集的并集，并且不重复。
UNION ALL	返回两个结果集的并集，并且可以重复。
INTERSECT	返回两个结果集的交集。
MINUS	返回两个结果集的差集。

集合运算符也有 规则和限制 。

连接（Join）

连接（Join）是将来自两个或多个表、视图或实例视图的行组合在一起的查询。 每当查询的 FROM 子句中出现多个表时，OceanBase 数据库执行连接。 查询的选择列表可以从其中任何表中选择任何列。 如果这两个表都有一个列名，那么您必须用表名限定查询过程中对这些列的所有引用。本节主要讲了以下连接：

连接类型	表示	说明
等值连接	EQUI-JOIN	包含等式运算符连接条件的连接。
自连接	SELF-JOIN	表与其自身的连接。
内连接	INNER JOIN	内连接，结果为两个连接表中的匹配行的连接。
左（外）连接	LEFT [OUTER] JOIN	结果包括左表（出现在 JOIN 子句最左边）中的所有行，不包括右表中的不匹配行。
右（外）连接	RIGHT [OUTER] JOIN	结果包括右表（出现在 JOIN 子句最右边）中的所有行，不包括有左表中的不匹配的行。
全（外）连接	FULL [OUTER] JOIN	结果包括所有连接中的所有行，不论他们是否匹配。
SEMI 连接	SEMI-JOIN	SEMI-JOIN 只能通过子查询展开得到。
ANTI 连接	ANTI-JOIN	ANTI-JOIN 也只能通过子查询展开得到。
笛卡儿积	Cartesian Products	当两个表没有连接操作时，对这两个表进行查询得到的数据是这两个表的笛卡儿积。

子查询

子查询指的是 SELECT 查询语句中嵌套了另一个或者多个 SELECT 语句，可以返回单行结果、多行结果或不返回结果。SELECT 语句的 FROM 子句中的子查询也称为内联视图。您可以在嵌入式视图中嵌套任意数量的子查询。SELECT 语句的 WHERE 子句中的子查询也称为嵌套子查询。您可以看 嵌套子查询的展开 和分布式查询。

9.2 简单查询

简单查询指从 OceanBase 一个或多个选择列表或视图中检索一个或多个列数据的操作，列的数量以及它们的数据类型和长度由选择列表的元素确定。而选择列表指的是 SELECT 关键字之后和 FROM 子句之前的表达式列表。

语法

```
SELECT 列名1,列名2,列名3,... FROM 表;
```

表名、字段名和关键字 SELECT、FROM 不区分大小写。查询的最后可以跟上分号（;），多条 SQL 可以同时执行。您可以使用 SELECT 语句中的 Hint 将指令或提示传递给 OceanBase 数据库优化器。优化器使用 Hint 为语句选择执行计划。

示例

创建一张员工表 **employee**，并向列 **employee_id**、**first_name**、**last_name**、**manager_id** 和 **salary** 插入数据：

```
CREATE TABLE employee (
employee_id INT
first_name VARCHAR(50),
last_name VARCHAR(50),
manager_id INT,
salary NUMERIC
);
INSERT INTO employee VALUES(111, 'DEL', 'FA BEN', 1, 1500);
INSERT INTO employee VALUES(112, 'AXEL', 'BELL', 1, 1000);
INSERT INTO employee VALUES(113, 'CRIS', 'RACHAR', 1, 1000);
```

简单查询示例

- 查询部分列：

```
SELECT first_name, last_name, salary FROM employee;
```

- 查询所有列：

```
SELECT * FROM employee;
```

- 对列进行数学运算：

```
SELECT salary+100 FROM employee;
```

- 给列取别名：

```
SELECT salary*12 年薪 FROM employee;
```

- 字符串拼接：

```
```

```
SELECT first_name || '-' || last_name AS 姓名 FROM employee;
```

- 数据去重：

```
SELECT DISTINCT 字段名 FROM 表名;
```

- CASE WHEN 语句：

```
SELECT salary, CASE WHEN salary >= 1000 then '高薪' WHEN salary >= 800 THEN '一般'
ELSE '继续努力' FROM employee;
```

9.3 层次查询

层次查询（Hierarchical Query）是一种具有特殊功能的查询语句，通过它能够将分层数据按照层次关系展示出来。分层数据是指关系表中的数据之间具有层次关系。这种关系在现实生活中十分常见，例如：

- 组织架构中组长和组员之间的关系。
- 企业中上下级部门之间的关系。
- Web 网页中，页面跳转的关系。

语法

```
SELECT [level], column, expr... FROM table [WHERE condition] [ START WITH start_expression ]
CONNECT BY [NOCYCLE] { PRIOR child_expr = parent_expr | parent_expr = PRIOR child_expr }
[ ORDER SIBLINGS BY ...] [ GROUP BY ... ] [ HAVING ... ] [ ORDER BY ... ]
```

参数

参数	说明
level	节点的层次，是伪列，表示等级。由查询的起点开始算起为 1，依次类推。
condition	条件。
CONNECT BY	指明如何来确定父子关系，这里通常使用等值表达式，但其他表达式同样支持。
START WITH	指明层次查询中的根行（Root row）。
PRIOR 运算符	PRIOR 是一元运算符，表示参数中的列来自于父行（Parent row），与一元的 + 和 - 具有相同的优先级。
NOCYCLE	当指定该关键字时，即使返回结果中有循环仍旧可以返回，并可以通过 CONNECT_BY_ISCYCLE 虚拟列来指明哪里出现循环；否则，出现循环会给客户端报错。
ORDER SIBLINGS BY	指定同一个层级行之间的排列顺序。

执行流程

使用和实现层次查询最关键是要理解其执行流程，层次查询执行流程：

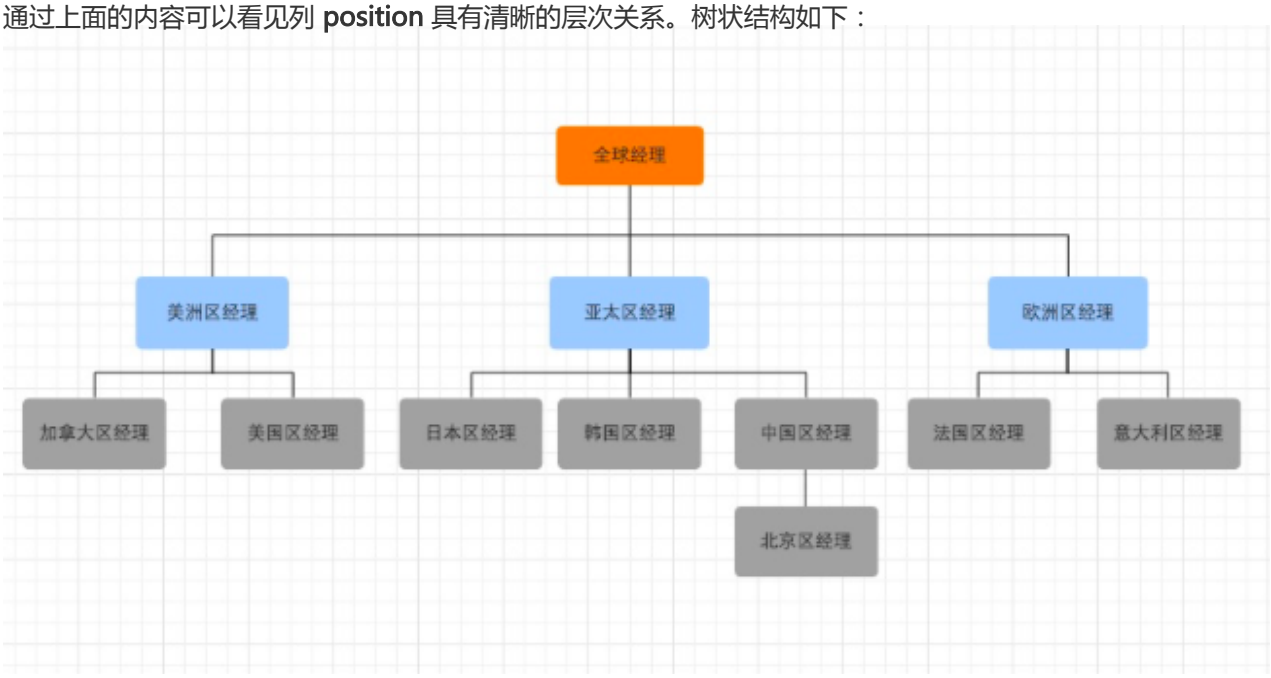
1. 执行 FROM 后面的 SCAN 或 JOIN 操作；
2. 根据 START WITH 和 CONNECT BY 的内容生成层次关系结果；
3. 按照常规查询执行流程执行剩下的子句（例如 WHERE、GROUP、ORDER BY……）对于 2 中生成层次关系的流程可以理解为：
4. 根据 START WITH 中的表达式得到根行（Root rows）。
5. 根据 CONNECT BY 中的表达式 选择每个根行（Root rows）的子行（Child rows）。
6. 将 2 中生成的子行（Child rows）作为新的根行（Root rows）进一步生成子行（Child rows），周而复始直到没有新行生成。

示例

展示层次查询的使用，向表 **emp** 中的 **emp_id**、**position** 和 **mgr_id** 列插入数据。执行以下语句：

```
CREATE TABLE emp(emp_id INT,position VARCHAR(50),mgr_id INT);
INSERT INTO emp VALUES (1,'全球经理',NULL);
INSERT INTO emp VALUES (2,'欧洲区经理',1);
INSERT INTO emp VALUES (3,'亚太区经理',1);
INSERT INTO emp VALUES (4,'美洲区经理',1);
INSERT INTO emp VALUES (5,'意大利区经理',2);
INSERT INTO emp VALUES (6,'法国区经理',2);
INSERT INTO emp VALUES (7,'中国区经理',3);
INSERT INTO emp VALUES (8,'韩国区经理',3);
INSERT INTO emp VALUES (9,'日本区经理',3);
INSERT INTO emp VALUES (10,'美国区经理',4);
INSERT INTO emp VALUES (11,'加拿大区经理',4);
INSERT INTO emp VALUES (12,'北京区经理',7);
```

通过上面的内容可以看见列 **position** 具有清晰的层次关系。树状结构如下：



是按照层次结构将结果展示出来，执行以下语句：

```
SELECT emp_id, mgr_id, position, level FROM emp
START WITH mgr_id IS NULL CONNECT BY PRIOR emp_id = mgr_id;
```

查询结果如下：

```
+--------+--------+------------------+-------+
| EMP_ID | MGR_ID | POSITION | LEVEL |
+--------+--------+------------------+-------+
| 1 | NULL | 全球经理 | 1 |
| 2 | 1 | 欧洲区经理 | 2 |
| 5 | 2 | 意大利区经理 | 3 |
| 6 | 2 | 法国区经理 | 3 |
| 3 | 1 | 亚太区经理 | 2 |
| 7 | 3 | 中国区经理 | 3 |
| 12 | 7 | 北京区经理 | 4 |
| 8 | 3 | 韩国区经理 | 3 |
| 9 | 3 | 日本区经理 | 3 |
| 4 | 1 | 美洲区经理 | 2 |
| 10 | 4 | 美国区经理 | 3 |
| 11 | 4 | 加拿大区经理 | 3 |
+--------+--------+------------------+-------+
```

如果仅查询亚太区的层次结构，执行以下语句：

```
SELECT emp_id, mgr_id, position, level FROM emp START WITH position = '亚太区经理' CONNECT BY PRIOR emp_id
= mgr_id;
```

查询结果如下：

```
+--------+--------+---------------+-------+
| EMP_ID | MGR_ID | POSITION | LEVEL |
+--------+--------+---------------+-------+
| 3 | 1 | 亚太区经理 | 1 |
| 7 | 3 | 中国区经理 | 2 |
| 12 | 7 | 北京区经理 | 3 |
| 8 | 3 | 韩国区经理 | 2 |
| 9 | 3 | 日本区经理 | 2 |
+--------+--------+---------------+-------+
```

9.4 集合

您可以使用集合运算符 UNION、UNION ALL、INTERSECT 和 MINUS 来组合多个查询。 所有集合运算符都具有相同的优先级。如果 SQL 语句包含多个集合运算符，则 OceanBase 从左到右对它们进行判断，除非括号中指定了顺序。

集合运算符的规则和限制

您可以指定 UNION 的属性为 ALL 和 DISTINCT 或 UNIQUE。分别代表集合可重复，和集合不可重复。而其它几种集合操作是不能指定 ALL 属性的（它们只有 DISTINCT 属性）。所有的集合操作默认的属性是 DISTINCT。在 Oceanbase 中，集合操作中可以指定 ORDER BY 和 LIMIT 子句，但是不允许其他子句的出现。我们仅 MINUS，不支持 EXCEPT 尽管这两者的语义是相同的。参加集合操作的各查询结果的列数和相应表达式的数量必须相同，对应的数据类型也必须兼容（例如数值或字符）。

集合运算符规则

如果组件查询是字符数据，则返回值的数据类型如下：

- 查询两个长度相等的 VARCHAR2 类型值，则返回的值为相同长度的 CHAR 类型。 查询不同长度的 CHAR 类型值，则返回的值为 VARCHAR2 类型，其长度为较大的 CHAR 值。
- 其中一个或两个查询是 VARCHAR2 数据类型的值，则返回的值为 VARCHAR2 类型。

如果组件查询是数值数据，则返回值的数据类型由数值优先级决定：

- 查询 BINARY_DOUBLE 类型值，则返回的值为 BINARY_DOUBLE 类型。
- 查询都选择 BINARY_FLOAT 类型值，则返回的值为 BINARY_FLOAT 类型。
- 所有查询都选择 NUMBER 类型值，则返回的值为 NUMBER 类型。

在使用集合运算符的查询中，OceanBase 不会跨数据类型组执行隐式转换。如果组件查询的相应表达式同时解析为字符数据和数值数据，OceanBase 将返回错误。

集合运算符限制

集合运算符受到以下限制：

- 集合运算符在 BLOB 和 CLOB 的列上无效。
- 如果集合运算符前面的 SELECT 列表包含表达式，则必须为表达式提供列别名，以便在 ORDER BY 子句引用。
- 您不能用设置的运算符指定 UPDATE 语句。
- 您不能在这些运算符的子查询中指定 ORDER BY 语句。
- 不能在包含表集合表达式的 SELECT 语句中使用这些运算符 。

UNION 运算符和 UNION ALL 运算符

UNION 运算符用于合并两个或多个 SELECT 语句的结果集。UNION 内部的 SELECT 语句必须拥有相同数量的列。列也必须拥有相似的数据类型。同时，每条 SELECT 语句中的列的顺序必须相同。默认地，UNION 运算符选取不同的值。如果允许重复的值，请使用 UNION ALL。

语法

```
{ (< SQL- 查询语句 1>) }
UNION [ALL]
{ (< SQL- 查询语句 2>) }
```

INTERSECT 运算符

返回两个结果集的交集，即两个查询都返回的所有非重复值。

语法

```
{ (< SQL- 查询语句 1>) }
INTERSECT
{ (< SQL- 查询语句 2>) }
```

限制条件

- 所有查询中的列数和列的顺序必须相同。
- 比较的两个查询结果集中的列数据类型可以不同但必须兼容。
- 比较的两个查询结果集中不能包含不可比较的数据类型列（XML、TEXT、NTEXT、IMAGE 或非二进制 CLR 用户定义类型）。
- 返回的结果集的列名与操作数左侧的查询返回的列名相同。ORDER BY 子句中的列名或别名必须引用左侧查询返回的列名。
- 不能与 COMPUTE 和 COMPUTE BY 子句一起使用。
- 通过比较行来确定非重复值时，两个 NULL 值被视为相等。

执行顺序

与表达式中的其他运算符一起使用时的执行顺序:
1、括号中的表达式。
2、INTERSECT 操作数。
3、基于在表达式中的位置从左到右求值的 EXCEPT 和 UNION。
如果 EXCEPT 或 INTERSECT 用于比较两个以上的查询集，则数据类型转换是通过一次比较两个查询来确定的，并遵循前面提到的表达式求值规则。

MINUS 运算符

MINUS 运算符返回两个结果集的差，即从左查询中返回右查询没有找到的所有非重复值。

语法

```
{ (< SQL- 查询语句 1>) }
MINUS
{ (< SQL- 查询语句 2>) }
```

示例

以下语句创建了表 **table_a** 和表 **table_b**，并向表中插入数据：

```
CREATE TABLE table_a(PK INT, name VARCHAR(25));
```

```
INSERT INTO table_a VALUES(1,'福克斯');
INSERT INTO table_a VALUES(2,'警察');
INSERT INTO table_a VALUES(3,'的士');
INSERT INTO table_a VALUES(4,'林肯');
INSERT INTO table_a VALUES(5,'纽约');
INSERT INTO table_a VALUES(6,'华盛顿');
INSERT INTO table_a VALUES(7,'戴尔');
INSERT INTO table_a VALUES(10,'朗讯');
CREATE TABLE table_b(PK INT, name VARCHAR(25));
INSERT INTO table_b VALUES(1,'福克斯');
INSERT INTO table_b VALUES(2,'警察');
INSERT INTO table_b VALUES(3,'的士');
INSERT INTO table_b VALUES(6,'华盛顿');
INSERT INTO table_b VALUES(7,'戴尔');
INSERT INTO table_b VALUES(8,'微软');
INSERT INTO table_b VALUES(9,'苹果');
INSERT INTO table_b VALUES(11,'苏格兰');
```

UNION 示例：：

```
SELECT PK, name FROM table_a
UNION
SELECT PK, name FROM table_b;
```

查询结果如下：

```
+------+-----------+
| PK | NAME |
+------+-----------+
| 1 | 福克斯 |
| 2 | 警察 |
| 3 | 的士 |
| 4 | 林肯 |
| 5 | 纽约 |
| 6 | 华盛顿 |
| 7 | 戴尔 |
| 10 | 朗讯 |
| 8 | 微软 |
| 9 | 苹果 |
| 11 | 苏格兰 |
+------+-----------+
```

UNION ALL 示例：：

```
SELECT PK, name FROM table_a
UNION ALL
SELECT PK, name FROM table_b;
```

查询结果如下：

```
+------+-----------+
```

```
| PK | NAME |
+------+-----------+
| 1 | 福克斯 |
| 2 | 警察 |
| 3 | 的士 |
| 4 | 林肯 |
| 5 | 纽约 |
| 6 | 华盛顿 |
| 7 | 戴尔 |
| 10 | 朗讯 |
| 1 | 福克斯 |
| 2 | 警察 |
| 3 | 的士 |
| 6 | 华盛顿 |
| 7 | 戴尔 |
| 8 | 微软 |
| 9 | 苹果 |
| 11 | 苏格兰 |
+------+-----------+
```

INTERSECT 示例：：

```
SELECT PK, NAME FROM table_a
INTERSECT
SELECT PK, NAME FROM table_b;
```

查询结果如下：

```
+------+-----------+
| PK | NAME |
+------+-----------+
| 1 | 福克斯 |
| 2 | 警察 |
| 3 | 的士 |
| 6 | 华盛顿 |
| 7 | 戴尔 |
+------+-----------+
```

MINUS 示例：

```
SELECT PK, NAME FROM table_a
MINUS
SELECT PK, NAME FROM table_b;
```

查询结果如下：

```
+------+--------+
| PK | NAME |
+------+--------+
| 4 | 林肯 |
| 5 | 纽约 |
```

```
| 10 | 朗讯 |
+------+--------+
```

9.5 连接

连接（Join）是将来自两个或多个表、视图或实例视图的行组合在一起的查询。 每当查询的 FROM 子句中出现多个表时，OceanBase 数据库执行连接。 查询的选择列表可以从其中任何表中选择任何列。 如果这两个表都有一个列名，那么您必须用表名限定查询过程中对这些列的所有引用。数据库中的连接类型一般包括 inner join、outer join、semi-join 和 anti-join。其中 Semi-join 和 Anti-join 都是通过子查询改写得到，OceanBase 本身并没有表述 Anti-join 和 Semi-join 的语法。

连接条件

连接条件可以分为等值连接（比如 t1.a = t2.b）和非等值连接（t1.a < t2.b）。相比于非等值连接条件, 等值连接条件的一个好处是允许数据库中使用高效的连接算法，比如 Hash Join 和 Merge-Sort join。

大多数连接查询在在从句或 WHERE 子句中至少包含一个连接条件。 连接条件比较两列，每列来自不同的表。为了执行连接，OceanBase 将每行组合成对，每个行包含每个表中的一行。 连接条件中的列也不需要出现在选择列表中。 要执行三个或多个表的连接，OceanBase 首先根据比较它们的列的连接条件连接其中的两个表，然后根据包含连接表和新表的列的连接条件将结果连接到另一个表。优化器根据连接条件、表上的索引以及表的任何可用统计数据确定 OceanBase 连接表的顺序。 包含连接条件的 WHERE 子句也可以包含其他条件，这些条件仅引用一个表的列。 这些条件可以进一步限制连接查询返回的行。 如果 WHERE 子句包含连接条件，则不能在 WHERE 子句中指定 LOB 列。

等值连接（EQUI-JOIN）

等值连接（Equijoins）是包含等式运算符连接条件的连接。等值连接组合了具有指定列的等效值的行。根据优化器选择执行连接的内部算法，单个表中 Equijoins 连接条件下的列的总大小可能仅限于数据块的大小减去一些开销。数据块的大小由初始化参数 DB_BLOCK_SIZE 指定。

自连接（SELF-JOIN）

自连接是表与其自身的连接。该表在 FROM 子句中出现两次，后跟表别名，这些别名限定连接条件中的列名。为了执行自连接，OceanBase 数据库将合并并返回满足连接条件的表行。

笛卡儿积（Cartesian Products）

如果连接查询中的两个表没有连接条件，则 OceanBase 数据库返回其笛卡尔乘积。 OceanBase 将一个表的每一行与另一行合并。笛卡尔乘积总是生成许多行，并且很少有用。 例如，两个都有 100 行的表的笛卡尔积有 10,000 行。除非您特别需要笛卡尔乘积，否则始终包括一个连接条件。 如果查询连接了三个或多个表，并且没有为特定对指定连接条件，则优化器可以选择避免生成中间笛卡尔乘积的连接顺序。

内连接（INNER JOIN）

内连接（INNER JOIN）是数据库中最基本的连接操作。内连接基于连接条件将两张表（如 A 和 B）的列组合在一起，产生新的结果表。查询会将 A 表的每一行和 B 表的每一行进行比较，并找出满足连接条件的组合。当连接条件被满足，A 和 B 中匹配的行会按列组合（并排组合）成结果集中的一行。连接产生的结果集，可以定义为首先对两张表做笛卡尔积（交叉连接） — 将 A 中的每一行和 B 中的每一行组合，然后返回满足连接条件的记录。

外连接（OUTER JOIN）

外连接（OUTER JOIN）扩展了简单连接的结果。 外部连接返回满足连接条件的所有行，并从一个表中返回部分或所有这些行，而另一个表中没有行满足连接条件。外连接可依据连接表保留左表、右表或全部表的行而进一步分为左连接、右连接和全连接。其中左连接（LEFT [OUTER] JOIN）中左表的一行未在右表中找到，就会在右表自动填充 NULL。右连接（RIGHT [OUTER] JOIN）中右表的一行未在左表中找到的时候，就在左表自动填充 NULL。全连接（FULL [OUTER] JOIN）就是左表或者右表找不匹配行的时候都会自动填充。

SEMI 连接（SEMI-JOIN）

当 A 表和 B 表进行 LEFT 或 RIGHT ANTI-JOIN 的时候，它只返回 A 或 B 中所有能够在 B 或 A 中找到匹配的行。SEMI-JOIN 只能通过子查询展开得到。

ANTI 连接（ANTI-JOIN）

当 A 表和 B 表进行 LEFT 或 RIGHT ANTI-JOIN 的时候，它只返回 A 或 B 中所有不能在 B 或 A 中找到匹配的行。类似于 SEMI-JOIN，ANTI-JOIN 也只能通过子查询展开得到。

示例

建立表 **table_a** 和表 **table_b**，并插入数据。执行以下语句：

```
CREATE TABLE table_a(PK INT, name VARCHAR(25));
INSERT INTO table_a VALUES(1,'福克斯');
INSERT INTO table_a VALUES(2,'警察');
INSERT INTO table_a VALUES(3,'的士');
INSERT INTO table_a VALUES(4,'林肯');
INSERT INTO table_a VALUES(5,'亚利桑那州');
INSERT INTO table_a VALUES(6,'华盛顿');
INSERT INTO table_a VALUES(7,'戴尔');
INSERT INTO table_a VALUES(10,'朗讯');
CREATE TABLE table_b(PK INT, name VARCHAR(25));
INSERT INTO table_b VALUES(1,'福克斯');
INSERT INTO table_b VALUES(2,'警察');
INSERT INTO table_b VALUES(3,'的士');
INSERT INTO table_b VALUES(6,'华盛顿');
INSERT INTO table_b VALUES(7,'戴尔');
INSERT INTO table_b VALUES(8,'微软');
INSERT INTO table_b VALUES(9,'苹果');
INSERT INTO table_b VALUES(11,'苏格兰威士忌');
```

自连接查询（SELF-JOIN）：

```
SELECT * FROM table_a ta, table_a tb WHERE ta.NAME = tb.NAME;
```

查询结果如下：

```
+------+----------------+------+----------------+
| PK | NAME | PK | NAME |
+------+----------------+------+----------------+
```

```
| 1 | 福克斯 | 1 | 福克斯 |
| 2 | 警察 | 2 | 警察 |
| 3 | 的士 | 3 | 的士 |
| 4 | 林肯 | 4 | 林肯 |
| 5 | 亚利桑那州 | 5 | 亚利桑那州 |
| 6 | 华盛顿 | 6 | 华盛顿 |
| 7 | 戴尔 | 7 | 戴尔 |
| 10 | 朗讯 | 10 | 朗讯 |
+------+----------------+------+----------------+
```

内连接查询（INNER JOIN）：

```
SELECT A.PK AS A_PK, A.name AS A_Value, B.PK AS B_PK, B.name AS B_Value
FROM table_a A INNER JOIN table_b B ON A.PK = B.PK;
```

查询结果如下：

```
+------+-----------+------+-----------+
| A_PK | A_VALUE | B_PK | B_VALUE |
+------+-----------+------+-----------+
| 1 | 福克斯 | 1 | 福克斯 |
| 2 | 警察 | 2 | 警察 |
| 3 | 的士 | 3 | 的士 |
| 6 | 华盛顿 | 6 | 华盛顿 |
| 7 | 戴尔 | 7 | 戴尔 |
+------+-----------+------+-----------+
```

左连接查询：

```
SELECT A.PK AS A_PK, A.name AS A_Value, B.PK AS B_PK, B.name AS B_Value
FROM table_a A LEFT JOIN table_b B ON A.PK = B.PK;
```

查询结果如下：

```
+------+----------------+------+-----------+
| A_PK | A_VALUE | B_PK | B_VALUE |
+------+----------------+------+-----------+
| 1 | 福克斯 | 1 | 福克斯 |
| 2 | 警察 | 2 | 警察 |
| 3 | 的士 | 3 | 的士 |
| 6 | 华盛顿 | 6 | 华盛顿 |
| 7 | 戴尔 | 7 | 戴尔 |
| 4 | 林肯 | NULL | NULL |
| 5 | 亚利桑那州 | NULL | NULL |
| 10 | 朗讯 | NULL | NULL |
+------+----------------+------+-----------+
```

右连接查询：

```
obclient> SELECT A.PK AS A_PK, A.name AS A_Value, B.PK AS B_PK, B.name AS B_Value FROM table_a A RIGHT JOIN
```

```
table_b B ON A.PK = B.PK;
```

查询结果如下：

```
+------+-----------+------+-------------------+
| A_PK | A_VALUE | B_PK | B_VALUE |
+------+-----------+------+-------------------+
| 1 | 福克斯 | 1 | 福克斯 |
| 2 | 警察 | 2 | 警察 |
| 3 | 的士 | 3 | 的士 |
| 6 | 华盛顿 | 6 | 华盛顿 |
| 7 | 戴尔 | 7 | 戴尔 |
| NULL | NULL | 8 | 微软 |
| NULL | NULL | 11 | 苏格兰威士忌 |
| NULL | NULL | 9 | 苹果 |
+------+-----------+------+-------------------+
```

全连接查询：

```
obclient> SELECT A.PK AS A_PK,A.name AS A_Value,B.PK AS B_PK,B.name AS B_Value FROM table_a A FULL JOIN
table_b B ON A.PK = B.PK;
```

查询结果如下：

```
+------+-----------------+------+-------------------+
| A_PK | A_VALUE | B_PK | B_VALUE |
+------+-----------------+------+-------------------+
| 1 | 福克斯 | 1 | 福克斯 |
| 2 | 警察 | 2 | 警察 |
| 3 | 的士 | 3 | 的士 |
| 6 | 华盛顿 | 6 | 华盛顿 |
| 7 | 戴尔 | 7 | 戴尔 |
| NULL | NULL | 8 | 微软 |
| NULL | NULL | 9 | 苹果 |
| NULL | NULL | 11 | 苏格兰威士忌 |
| 4 | 林肯 | NULL | NULL |
| 5 | 亚利桑那州 | NULL | NULL |
| 10 | 朗讯 | NULL | NULL |
+------+-----------------+------+-------------------+
```

Semi 连接查询（semi-join）：有依赖关系的子查询被展开改写成 **SEMI-JOIN**。

```
explain SELECT * FROM table_a t1 WHERE t1.PK IN (SELECT t2.PK FROM table_b t2 WHERE t2.NAME = t1.NAME);
```

查询结果如下：

```
+-----------------------------------------------+
| Query Plan |
+-----------------------------------------------+
=========================================
```

```
|ID|OPERATOR |NAME|EST. ROWS|COST|
--------------------------------------
|0 |HASH SEMI JOIN| |8 |114 |
|1 | TABLE SCAN |T1 |8 |38 |
|2 | TABLE SCAN |T2 |8 |38 |
=======================================

Outputs & filters:
-------------------------------------
0 - output([T1.PK], [T1.NAME]), filter(nil),
equal_conds([T1.PK = T2.PK], [T2.NAME = T1.NAME]), other_conds(nil)
1 - output([T1.NAME], [T1.PK]), filter(nil),
access([T1.NAME], [T1.PK]), partitions(p0)
2 - output([T2.NAME], [T2.PK]), filter(nil),
access([T2.NAME], [T2.PK]), partitions(p0)
+---------------------------------------------+
```

Anti 连接查询（anti-join）：有依赖关系的子查询被改写成 Anti-join。

```
EXPLAIN SELECT * FROM table_a t1 WHERE t1.PK NOT IN (SELECT t2.PK
FROM table_b t2 WHERE t2.name = t1.name);
```

查询结果如下：

```
+---------------------------------------------+
| Query Plan |
+---------------------------------------------+
=======================================
|ID|OPERATOR |NAME|EST. ROWS|COST|
--------------------------------------
|0 |HASH ANTI JOIN| |0 |112 |
|1 | TABLE SCAN |T1 |8 |38 |
|2 | TABLE SCAN |T2 |8 |38 |
=======================================
Outputs & filters:
-------------------------------------
0 - output([T1.PK], [T1.NAME]), filter(nil),
equal_conds([T2.NAME = T1.NAME]), other_conds([(T_OP_OR, T1.PK = T2.PK,
(T_OP_IS, T1.PK, NULL, 0), (T_OP_IS, T2.PK, NULL, 0))])
1 - output([T1.NAME], [T1.PK]), filter(nil),
access([T1.NAME], [T1.PK]), partitions(p0)
2 - output([T2.NAME], [T2.PK]), filter(nil),
access([T2.NAME], [T2.PK]), partitions(p0)
+-----------------------------------------------------+
```

9.6 子查询

子查询指的是 SELECT 查询语句中嵌套了另一个或者多个 SELECT 语句，可以返回单行结果、多行结果或不返回结果。SELECT 语句的 FROM 子句中的子查询也称为内联视图。您可以在嵌入式视图中嵌套任意数量的子查询。SELECT 语句的 WHERE 子句中的子查询也称为嵌套子查询。

子查询可以分成有依赖关系的子查询和没有依赖关系的子查询。有依赖关系的子查询是指该子查询的执行依赖

了外部查询的变量，所以这种子查询通常会被计算多次。没有依赖关系的子查询是指该子查询的执行不依赖外部查询的变量，这种子查询一般只需要计算一次。子查询按类型可分为：

- 单行子查询，不向外部返回结果，或者只返回一行结果。
- 多行子查询，向外部返回零行、一行或者多行结果。

语法

```
SELECT [ hint ] [ { { DISTINCT | UNIQUE } | ALL } ] select_list
FROM { table_reference | join_clause | ( join_clause ) }
[ , { table_reference | join_clause | (join_clause) } ]
[ where_clause ]
[ hierarchical_query_clause ]
[ group_by_clause ]
| subquery { UNION [ALL] | INTERSECT | MINUS } subquery [ { UNION [ALL] | INTERSECT | MINUS } subquery ]
| ( subquery ) [ order_by_clause ] [ row_limiting_clause ]
```

参数

参数	说明
select_list	查询列表。
subquery	子查询。
hint	注释。
table_reference	要查询的目标表。

如果子查询中的列与包含语句中的列具有相同的名称，则必须在包含语句中对表列的任何引用之前加上表名或别名。您可以将子查询中的列与表、视图或实例视图的名称或别名进行限定。

当嵌套子查询引用来自引用到子查询上方一个级别的父语句的表的列时，OceanBase 执行相关的子查询。父语句可以是嵌套子查询的选择、更新或删除语句。Hint 可以选择重写查询作为连接，或者使用其他技术来制定语义等效的查询。OceanBase 通过查看子查询中命名的表，然后在父语句中命名的表中查找子查询中的不合格列，从而解析子查询中的不合格列。

当嵌套的子查询引用表中的列时，OceanBase 会执行相关的子查询，该表引用了子查询上一级的父语句。父查询语句可以是子查询嵌套在其中的 SELECT，UPDATE 或 DELETE 语句。从概念上讲，对关联的子查询对父查询语句处理的每一行都进行一次评估。但是，Hint 可以选择将查询重写为联接，也可以使用其他某种技术来制定语义上等效的查询。OceanBase 通过在子查询中命名的表然后在父语句中命名的表中查找子查询中的非限定列。相关子查询回答一个多部分问题，其答案取决于父语句处理的每一行中的值。

以下语句会用到子查询:

- 定义要插入到 INSERT 或 CREATE TABLE 语句的目标表中的行集。
- 在 CREATE VIEW 或 CREATE MATERIALIZED VIEW 语句中定义要包含在视图或实例化视图中的行集。
- 在 UPDATE 中定义要分配给现有行的一个或多个值。
- 在 WHERE 子句，HAVING 子句或 START WITH 中提供条件值 SELECT、UPDATE 和 DELETE 语句的子句。
- 定义包含查询操作的表。

嵌套子查询的展开（Unnesting of Nested Subqueries）

嵌套子查询展开是数据库的一种优化策略，它把一些子查询置于外层的父查询中，其实质是把某些子查询转化为等价的多表连接操作。这种策略带来的一个明显的好处是写访问路径、连接方法和连接顺序可能被有效的利用，使得查询语句的层次尽可能的减少。

Hint 将自动取消以下嵌套子查询:

- 不相关的 IN 子查询。
- 只要 IN 和 EXISTS 相关子查询不包含聚合函数或 GROUP BY 子句。
- 您可以通过指示 Hint 来启用扩展子查询。
- 您可以通过指定 HASH_AJ 来取消嵌套不相关的 NOT IN 子查询中的 MERGE_AJ 提示。
- 您可以通过在子查询中指定 UNNEST 提示来取消嵌套其他子查询。

分布式查询

您可以使用 OceanBase 客户端远程访问 OceanBase 分布式数据库里的数据。

示例

以下语句创建了表 **table_a**和表 **table_b**，并向表中插入数据：

```
CREATE TABLE table_a(PK INT, name VARCHAR(25));
INSERT INTO table_a VALUES(1,'福克斯');
INSERT INTO table_a VALUES(2,'警察');
INSERT INTO table_a VALUES(3,'的士');
INSERT INTO table_a VALUES(4,'林肯');
INSERT INTO table_a VALUES(5,'亚利桑那州');
INSERT INTO table_a VALUES(6,'华盛顿');
INSERT INTO table_a VALUES(7,'戴尔');
INSERT INTO table_a VALUES(10,'朗讯');
CREATE TABLE table_b(PK INT, name VARCHAR(25));
INSERT INTO table_b VALUES(1,'福克斯');
INSERT INTO table_b VALUES(2,'警察');
INSERT INTO table_b VALUES(3,'的士');
INSERT INTO table_b VALUES(6,'华盛顿');
INSERT INTO table_b VALUES(7,'戴尔');
INSERT INTO table_b VALUES(8,'微软');
INSERT INTO table_b VALUES(9,'苹果');
INSERT INTO table_b VALUES(11,'苏格兰威士忌');
```

没有依赖关系的子查询，执行以下语句：

```
SELECT * FROM TABLE_A T1 WHERE T1.PK IN (SELECT T2.PK FROM TABLE_B T2);
```

查询结果如下：

```
+------+-----------+
```

```
| PK | NAME |
+------+-----------+
| 1 | 福克斯 |
| 2 | 警察 |
| 3 | 的士 |
| 6 | 华盛顿 |
| 7 | 戴尔 |
+------+-----------+
```

有依赖关系的子查询，子查询中用到了外层查询变量 **T1.PK**，执行以下语句：

```
SELECT * FROM TABLE_A T1 WHERE T1.PK IN (SELECT T2.PK FROM TABLE_B T2 WHERE T2.PK = T1.PK);
```

查询结果如下：

```
+------+-----------+
| PK | NAME |
+------+-----------+
| 1 | 福克斯 |
| 2 | 警察 |
| 3 | 的士 |
| 6 | 华盛顿 |
| 7 | 戴尔 |
+------+-----------+
```

有依赖关系的子查询被展开改写成连接，执行以下语句：

```
EXPLAIN SELECT * FROM TABLE_A T1 WHERE T1.PK IN (SELECT T2.NAME FROM TABLE_B T2 WHERE T2.NAME =
T1.NAME);
```

查询结果如下：

```
+------------------------------------+
| Query Plan |
+------------------------------------+
============================================
|ID|OPERATOR |NAME|EST. ROWS|COST|
--------------------------------------------
|0 |HASH RIGHT SEMI JOIN| |8 |107 |
|1 | TABLE SCAN |T2 |8 |38 |
|2 | TABLE SCAN |T1 |8 |38 |
============================================
Outputs & filters:
-------------------------------------
0 - output([T1.PK], [T1.NAME]), filter(nil),
equal_conds([T1.PK = T2.NAME], [T2.NAME = T1.NAME]), other_conds(nil)
1 - output([T2.NAME]), filter(nil),
access([T2.NAME]), partitions(p0)
2 - output([T1.NAME], [T1.PK]), filter(nil),
access([T1.NAME], [T1.PK]), partitions(p0)
+------------------------------------+
```


蚂蚁金服
ANT FINANCIAL
金融科技
FINANCIAL TECHNOLOGY