第2章 关系数据模型与SQL的发展

大数据管理概论

第2章 关系数据模型与SQL

- **教学内容**
  - 本章讲述关系数据模型与SQL语言的基础知识与概念；
  - 面向大数据管理需求的数据库实现技术及SQL扩展技术；
  - 通过代表性数据库分析介绍关系数据库的主要实现技术。
- **教学目标**
  1）能够阐述关系数据库的基本概念、操作与实现技术；
  2）能够阐释SQL基本语法和扩展语法的特征；
  3）能够描述SQL on Hadoop的典型应用案例；
  4）能够列举并陈述NoSQL数据库的特点；
  5）能够列举并陈述代表性的关系、MPP、NewSQL数据库。

第2章 关系数据模型与SQL

2.1 关系数据库概述

2.2 关系数据库标准语言SQL

- SQL for XML
- SQL for JSON

2.3 SQL on Hadoop

2.4 NoSQL数据库 vs. 关系数据库

2.5 代表性数据库演化与发展趋势

2.1关系数据库概述
2.1.1关系数据结构及其形式化定义

单一的数据结构-关系

现实世界的实体以及实体间的各种联系均用关系来表示。

关系代数

通过代数方式执行关系操作，以关系为运算对象和运算结果。传统的集合运算和专门的关系运算(选择、投影、连接、除等)两大类。

关系的基本概念

关系、元组、属性、主属性、非主属性、域、主码、外码、...

关系的特征

二维表：行、列，满足1NF。关系模式：预先定义，按关系模式组织数据；不适合稀疏存储与无模式数据存储。

形式化描述R(U,D,DOM,F)

函数依赖：关系范式2NF、3NF、BCNF。

数据依赖：关系范式4NF。

2.1.1 关系数据结构及其形式化定义

- **关系数据库主要优点：**

1.完整的模式定义

2.较高的存储效率

3.完善的ACID特性

4.支持复杂查询——连接、嵌套

5.较高的查询处理性能——索引、查询优化

6.扩展性——纵向提升机器性能，横向MPP受到一致性束缚

7.易用性

8.丰富的开发工具

- **关系数据库主要缺点：**

1.不适合无模式及模式动态变化的应用场景

2.写入性能问题——事务延迟、分布式数据库

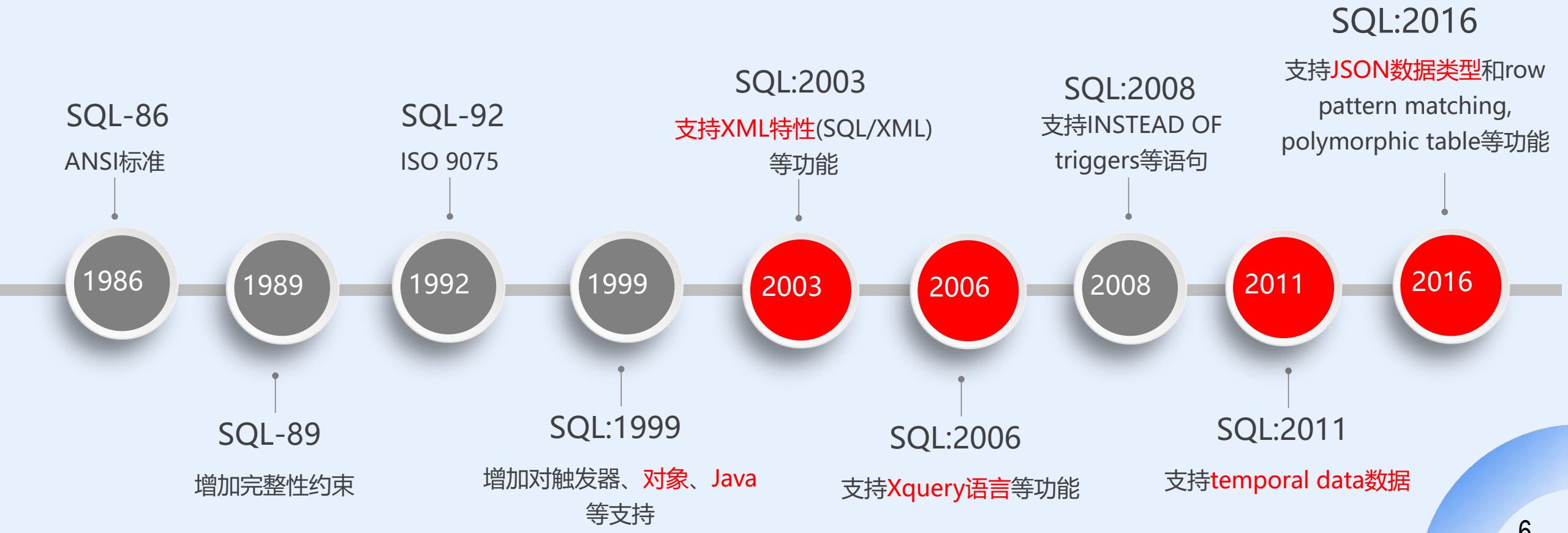
3.扩展性较低——大规模集群受ACID束缚

4.简单查询未必快速返回结果——解析、并发控制

5.非结构化数据管理能力不足——文档、XML文件、JSON

2.2 关系数据库标准语言SQL
2.2.1 基本SQL标准

- SQL语言：高度非过程化，面向集合操作的数据库查询语言
- SQL标准演变：扩展对新数据类型及功能的支持



2.2.1 关系数据库标准语言SQL（续）

- **数据定义**

操作对象	操作方式		
	创建	删除	修改
模式	CREATE SCHEMA	DROP SCHEMA	ALTER SCHEMA
表	CREATE TABLE	DROP TABLE	ALTER TABLE
索引	CREATE INDEX	DROP INDEX	ALTER INDEX
视图	CREATE VIEW	DROP VIEW	ALTER VIEW

- **数据查询**
  - **基本查询**:选择、投影、连接、分组、聚集、排序等
  - **扩展查询**: 面向OLAP分析的聚合计算
- **连接操作的实现机理**
  - 嵌套循环、排序归并、哈希连接
- **数据更新**
  - 增、删、改操作
  - 更新的约束：完整性约束条件（实体、参照、用户定义）

2.2 关系数据库标准语言SQL
2.2.2 面向大数据管理的SQL扩展语法

- **SQL for XML**

Select 的查询结果会作为行集返回，可以在sql中指定for xml子句使得查询结果转换为xml。有4种模式：

模式名	功能
RAW	返回的行作为元素，列值作为元素的属性
AUTO	返回表名对应节点名称的元素，每列的属性作为元素的属性输出，可形成简单嵌套结构
EXPLICIT	通过SELECT语法定义输出XML结构
PATH	列名或列别名作为XPATH表达式来处理

JSON 语法规则

- JSON 语法是 JavaScript 对象表示语法的子集。
  - 数据在名称/值对中，
  - JSON 值可以是：
    - 数字（整数或浮点数）
    - 字符串（在双引号中）
    - 逻辑值（true 或 false）
    - **数组（在中括号中）**
    - **对象（在大括号中，KV对）**
    - null
- 数据由逗号分隔

- JSON 数组：在中括号 [] 中书写：
  - 数组可包含多个对象（每个对象在大括号{ }中书写），例如：

```
{ "sites":
  [ { "name":"菜鸟教程" ,
"url":"www.runoob.com" },
    { "name":"google" ,
"url":"www.google.com" },
    { "name":"微博" ,
"url":"www.weibo.com" }
  ]
}
```

XML vs. JSON语法差异

- XML示例：

```
<?xml version="1.0" encoding="utf-8" ?>
<country>
  <name>中国</name>
  <province>
    <name>黑龙江</name>
    <citys>
      <city>哈尔滨</city>
      <city>大庆</city>
    </citys>
  </province>
  <province>
    <name>广东</name>
    <citys>
      <city>广州</city>
      <city>深圳</city>
      <city>珠海</city>
    </citys>
  </province>
……
  </province>
```

- JSON示例：

```
var country =
      {
         name: "中国",
         provinces: [
         { name: "黑龙江", citys: { city: ["哈尔滨", "大庆"]} },
         { name: "广东", citys: { city: ["广州", "深圳", "珠海"]} },
        ……
         ]
      }
```

- **标量(scalar)**：单一的数字、bool、string、null都可以叫做标量。
- **数组(array)**：[]结构，里面存放的元素可以是任意类型的JSON，并且不要求数组内所有元素都是同一类型。
- **对象(object)**：{}结构，存储key:value的键值对，其键只能是用“”包裹起来的字符串，值可以是任意类型的JSON，对于重复的键，按最后一个键值对为准。

SQL for XML, RAW

```
CREATE TABLE [dbo].[Users](
    [UserID] [int] IDENTITY(1,1) NOT NULL,
    [FirstName] [nvarchar](50) NULL,
    [LastName] [nvarchar](50) NULL
)
```

- **SELECT UserID, FirstName,LastName FROM users FOR XML RAW ('MyUsers');**

说明：返回的每行均是一个元素，列值作为元素的属性；将**元素命名为自定义的名称**。

```
<MyUsers UserID="1" FirstName="HSQOFNPSCF" LastName="DEVGNBDCTQ" />
<MyUsers UserID="2" FirstName="IALXYEHSTB" LastName="IGSVYHGMJT" />
<MyUsers UserID="3" FirstName="NLMVBIECDU" LastName="LQOWILPIAH" />
<MyUsers UserID="4" FirstName="IFYIIJCJCC" LastName="SSEEQNMLTF" />
<MyUsers UserID="5" FirstName="XEVKGGSCND" LastName="QDWOPBGLHM" />
<MyUsers UserID="6" FirstName="YJUPQHJSLC" LastName="SMKGOPALPA" />
<MyUsers UserID="7" FirstName="LXRXLJPKAL" LastName="TRQSAQSUDV" />
<MyUsers UserID="8" FirstName="SNYRTVADDP" LastName="UYTMIHVGYE" />
<MyUsers UserID="9" FirstName="EAMUTPEYDR" LastName="MPTYUFDLAF" />
<MyUsers UserID="10" FirstName="PORKGOIPBV" LastName="HBPRUJEBAR" />
```

结果 消息

	UserID	FirstName	LastName
1	1	HSQOFNPSCF	DEVGNBDCTQ
2	2	IALXYEHSTB	IGSVYHGMJT
3	3	NLMVBIECDU	LQOWILPIAH
4	4	IFYIIJCJCC	SSEEQNMLTF
5	5	XEVKGGSCND	QDWOPBGLHM
6	6	YJUPQHJSLC	SMKGOPALPA
7	7	LXRXLJPKAL	TRQSAQSUDV
8	8	SNYRTVADDP	UYTMIHVGYE
9	9	EAMUTPEYDR	MPTYUFDLAF
10	10	PORKGOIPBV	HBPRUJEBAR

SQL for XML, AUTO

- SELECT UserID, FirstName,LastName FROM users **FOR XML AUTO, XMLSCHEMA**;

说明：以表名为名称的元素，每列的属性作为属性输出；加上**XMLSCHEMA则输出xml架构**，不加则只输出数据。

```
CREATE TABLE [dbo].[Users](
   [UserID] [int] IDENTITY(1,1) NOT NULL,
   [FirstName] [nvarchar](50) NULL,
   [LastName] [nvarchar](50) NULL
)
```

```
<users UserID="1" FirstName="HSQOFNPSCF" LastName="DEVGNBDCTQ" />
<users UserID="2" FirstName="IALXYEHSTB" LastName="IGSVYHGMJT" />
<users UserID="3" FirstName="NLMVBIECDU" LastName="LQOWILPIAH" />
<users UserID="4" FirstName="IFYIIJCJCC" LastName="SSEEQNMLTF" />
<users UserID="5" FirstName="XEVKGGSCND" LastName="QDWOPBGLHM" />
<users UserID="6" FirstName="YJUPQHJSLC" LastName="SMKGOPALPA" />
<users UserID="7" FirstName="LXRXLJPKAL" LastName="TRQSAQSUDV" />
<users UserID="8" FirstName="SNYRTVADDP" LastName="UYTMIHVGYE" />
<users UserID="9" FirstName="EAMUTPEYDR" LastName="MPTYUFDLAF" />
<users UserID="10" FirstName="PORKGOIPBV" LastName="HBPRUJEBAR" />
```

结果 消息

	UserID	FirstName	LastName
1	1	HSQOFNPSCF	DEVGNBDCTQ
2	2	IALXYEHSTB	IGSVYHGMJT
3	3	NLMVBIECDU	LQOWILPIAH
4	4	IFYIIJCJCC	SSEEQNMLTF
5	5	XEVKGGSCND	QDWOPBGLHM
6	6	YJUPQHJSLC	SMKGOPALPA
7	7	LXRXLJPKAL	TRQSAQSUDV
8	8	SNYRTVADDP	UYTMIHVGYE
9	9	EAMUTPEYDR	MPTYUFDLAF
10	10	PORKGOIPBV	HBPRUJEBAR

SQL for XML, AUTO

- Raw和auto的区别： auto可以形成简单的层次关系，表名作为节点，多表连接时从左至右的表依次形成父子关系的嵌套结构。

select student.id, student.name,teacher.teacherId,teacher.teachername from student inner join teacher on student.teacherId=teacher.teacherId for xml raw;

```
<row id="10" name="小李                " teacherId="1" teacherName="王静" />
<row id="11" name="小方                " teacherId="2" teacherName="李四" />
```

select student.id, student.name,teacher.teacherId,teacher.teachername from student inner join teacher on student.teacherId=teacher.teacherId for xml auto;

```
/*  生成了嵌套关系
<student id="10" name="小李                ">
  <teacher teacherId="1" teacherName="王静" />
</student>
<student id="11" name="小方                ">
  <teacher teacherId="2" teacherName="李四" />
</student>
*/
```

SQL for XML, PATH

- SELECT UserID "@ID", FirstName "Name/FirstName", LastName "Name/LastName" FROM users FOR XML PATH ('MyUsers');

说明：列名或列别名作为XPATH表达式来处理；可以指定xml结构。

结果 消息

	UserID	FirstName	LastName
1	1	HSQOFNPSCF	DEVGNBDCTQ
2	2	IALXYEHSTB	IGSVYHGMJT
3	3	NLMVBIECDU	LQOWILPIAH
4	4	IFYIIJCJCC	SSEEQNMLTF
5	5	XEVKGGSCND	QDWOPBGLHM
6	6	YJUPQHJSLC	SMKGOPALPA
7	7	LXRXLJPKAL	TRQSAQSUDV
8	8	SNYRTVADDP	UYTMIHVGYE
9	9	EAMUTPEYDR	MPTYUFDLAF
10	10	PORKGOIPBV	HBPRUJEBAR

```
<MyUsers ID="1">
  <Name>
    <FirstName>HSQOFNPSCF</FirstName>
    <LastName>DEVGNBDCTQ</LastName>
  </Name>
</MyUsers>
<MyUsers ID="2">
  <Name>
    <FirstName>IALXYEHSTB</FirstName>
    <LastName>IGSVYHGMJT</LastName>
  </Name>
</MyUsers>
<MyUsers ID="3">
  <Name>
    <FirstName>NLMVBIECDU</FirstName>
    <LastName>LQOWILPIAH</LastName>
  </Name>
</MyUsers>
<MyUsers ID="4">
  <Name>
    <FirstName>IFYIIJCJCC</FirstName>
    <LastName>SSEEQNMLTF</LastName>
  </Name>
</MyUsers>
<MyUsers ID="5">
  <Name>
    <FirstName>XEVKGGSCND</FirstName>
    <LastName>QDWOPBGLHM</LastName>
  </Name>
</MyUsers>
```

SQL for XML, EXPLICIT 通过SELECT语法定义输出XML结构

编写SELECT查询以生成具有特定格式的行集（通用表），并对应生成所需的 XML结构。

- 语法规则-元数据列

SELECT查询必须先生成满足规定要求的前两列，即Tag列和Parent列，称为元数据列，作用是为结果提供层次信息：

- 第1列，列名固定为Tag，值是一个对应当前元素的标记号（整数类型）。 查询必须为从行集构造的每个元素提供标记号。
- 第2列，列名固定为Parent，值则是父元素的标记号。Parent 列值为0或NULL表明相应的元素没有父级，该元素将作为顶级元素添加到 XML。

元数据列值与列名中的信息一起用于生成所需的 XML。 查询必须以特定方式提供列名。

参考链接：https://learn.microsoft.com/zh-cn/sql/relational-databases/xml/use-explicit-mode-with-for-xml?view=sql-server-ver16

SQL for XML, EXPLICIT　　通过SELECT语法定义输出XML结构

- 语法规则-其他数据列名的指定方式

  ElementName!TagNumber!AttributeName!Directive

  其中Directive 是可选的，提供有关 XML 构造的其他信息，有两种用途：

  （1）将值编码为ID、IDREF 和IDREFS关键字，从而支持创建文档内链接。

  （2）用来指示如何将字符串数据映射到 XML，可以将hide、element、elementxsinil、xml、xmltext和cdata关键字用作Directive。

在从通用表生成 XML 的过程中，表中的数据被垂直分区到列组中。分组是根据 Tag 值和列名确定的，每行中相同Tag值和包含该Tag值的列名为一列组，相应构造一个XML元素。

参考链接：https://learn.microsoft.com/zh-cn/sql/relational-databases/xml/use-explicit-mode-with-for-xml?view=sql-server-ver16

SQL for XML, EXPLICIT　　通过SELECT语法定义输出XML结构

- 例1：假定生成的通用表如下：

Tag	Parent	Customer!1!cid	Customer!1!name	Order!2!id	Order!2!date	OrderDetail!3!id!id	OrderDetail!3!pid!idref
1	NULL	C1	"Janine"	NULL	NULL	NULL	NULL
2	1	C1	NULL	01	1/20/1996	NULL	NULL
3	2	C1	NULL	01	NULL	OD1	P1
3	2	C1	NULL	01	NULL	OD2	P2
2	1	C1	NULL	02	3/29/1997	NULL	NULL

```
<Customer cid="C1" name="Janine">
    <Order id="01" date="1/20/1996">
        <OrderDetail id="OD1" pid="P1"/>
        <OrderDetail id="OD2" pid="P2"/>
    </Order>
    <Order id="02" date="3/29/1997">
</Customer>
```

- 应用列组的生成规则：
  - 对于**第一行中的 Tag 列值 1**，名称中包括此相同标记号的列（Customer!1!cid 和 Customer!1!name）形成一组。
  - 对于 **Tag 列值为 2 的行**，列 Order!2!id 和 Order!2!date 构成一个组，<Order id=... date=... />，用于构造元素 <Order id=... date=... >。
  - 对于 **Tag 列值为 3 的行**，列 OrderDetail!3!id!id 和OrderDetail!3!pid!idref 形成一组。这些行中的每一行都从这些列生成一个元素 <OrderDetail id=... pid=...>。

参考链接：https://learn.microsoft.com/zh-cn/sql/relational-databases/xml/use-explicit-mode-with-for-xml?view=sql-server-ver16

SQL for XML, EXPLICIT　　通过SELECT语法定义输出XML结构

例2：SELECT TOP 5 1 AS Tag, 0 AS Parent, OrderID AS [Order!1!ID],
OrderDate AS [Order!1!Date],CustomerID AS [Order!1!Customer],NULL AS
[OrderDetail!2!ProductID],NULL AS [OrderDetail!2!UnitPrice],NULL AS [OrderDetail!2!Quantity]
FROM dbo.Orders WHERE dbo.Orders.OrderID='10248'
UNION ALL
SELECT TOP 5 2 AS Tag, 1 AS Parent, NULL,NULL,NULL,ProductID,UnitPrice,Quantity
FROM dbo.[Order Details] WHERE OrderID='10248' FOR XML EXPLICIT

查询结果：

```
<Order ID="10248" Date="1996-07-04T00:00:00" Customer="VINET">
  <OrderDetail ProductID="11" UnitPrice="14.0000" Quantity="12" />
  <OrderDetail ProductID="42" UnitPrice="9.8000" Quantity="10" />
  <OrderDetail ProductID="72" UnitPrice="34.8000" Quantity="5" />
</Order>
```

2.2 关系数据库标准语言SQL
2.2.2 面向大数据管理的SQL扩展语法（续）

- **SQL for JSON数据管理**

  JSON是一种轻量级的数据交换格式，采用完全独立于编程语言的文本格式来存储和表示数据。

  - SQL:2016标准中增加了对JSON数据结构的支持。
  - Oracle 12c、MySQL 5.7、SQL Server 2016等数据库增加了对JSON的数据管理功能，通过内置接口支持对JSON的存储、解析、查询、索引等功能。

SQL for JSON数据管理

- SQL Server DBMS中典型的JSON数据管理功能接口。支持：
  - 分析 JSON 文本和读取或修改值。
  - 将 JSON 对象数组转换为关系格式。
  - 在转换后的 JSON 对象上运行任意 Transact-SQL 查询。
  - 将 Transact-SQL 查询的结果设置为 JSON 格式。

功能	描述
OPENJSON	解析JSON数据
ISJSON	测试字符串是否包含有效的JSON
JSON_VALUE	从JSON字符串中提取标量值
JSON_QUERY	从JSON字符串中提取对象或数组
JSON_MODIFY	更新JSON字符串中的属性值，并返回更新的JSON字符串
JSON_OBJECTAGG	通过聚合 SQL 数据或列来构造 JSON 对象
JSON_ARRAYAGG	通过聚合 SQL 数据或列来构造 JSON 数组

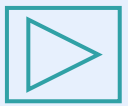
https://learn.microsoft.com/zh-cn/sql/relational-databases/json/json-data-sql-server?view=sql-server-ver16

SQL for JSON数据管理

阅读教材 P26-29 例题：

1. 解析JSON数据 (**OPENJSON...WITH**)
2. JSON数据转换为关系数据 (**SELECT ...INTO tab FROM OPENJSON...WITH**)
3. JSON数据更新为关系数据列

   (**UPDATE tab SET json_col=@json Variable..., JSON_MODIFY**)
4. SQL查询中使用关系和JSON数据 (**JSON_VALUE,ISJSON,CROSS APPLY OPENJSON**)
5. JSON索引 (**虚拟列上的索引**)
6. 关系数据输出为JSON数据格式 (**SELECT ...FOR JSON AUTO|PATH**)



SQL for JSON数据管理

1. 解析JSON数据

```
DECLARE @json Variable NVARCHAR(MAX)
SET @json Variable=N'[...]'          --加上N代表存入数据库时以Unicode格式存储
...
SELECT * FROM OPENJSON(@json Variable)  --OPENJSON解析JSON数据
WITH (id int  'strict $.id',...);        --WITH设置JSON数据解析结构strict表示json中必须包含该字段
```

2. JSON数据转换为关系数据

```
SELECT * into region_json
FROM OPENJSON(@json Variable)  --将解析出的JSON数据插入表中
WITH (id int  'strict $.id',...);          --名称/值转化为属性的值
```

SQL for JSON数据管理

3. JSON数据更新到关系数据列

```
DECLARE @json Variable0 NVARCHAR(MAX);
SET @json Variable0='{"id":0,
"Location":{"Horizontal_region":"East","Vertical_region":"South"},...}'
DECLARE @json Variable1 NVARCHAR(MAX);
SET @json Variable1='{"id":1, "Location":{...}...}';
SET @json = JSON_MODIFY(@json, '$.Location.Horizontal_region', 'North') ;...
Update REGION set json_col=@json Variable0 where r_key=0;
Update REGION set json_col=@json Variable1 where r_key=1;
...
SELECT r_name,
JSON_VALUE(json_col,'$.Location.Horizontal_region') AS Loca_H,
JSON_VALUE(json_col,'$.Location. Vertical_region') AS Loca_V FROM REGION;
```

--JSON_VALUE用于从JSON字符串中解析值

SQL for JSON数据管理

4. SQL查询中使用关系和JSON数据

```
SELECT
FROM REGION AS R
        CROSS APPLY
        OPENJSON(R.json_col) WITH(...) AS Detail
WHERE ISJSON(json_col)>0 AND Detail.People>0.8
--OPENJSON用于将JSON数据转换为关系数据格式，用Json表达式用于不同查询子句
```

cross apply左部关系的每一行都和派生表（表值函数根据R当前行数据生成的动态结果集）做一个交叉联接（cross join）。

参见：教材28页例2-6

SQL for JSON数据管理

5. JSON索引

```
ALTER TABLE R1
  ADD vHorizontal_region AS JSON_VALUE(json_col, '$.Location.Horizontal_region');
--创建一个json属性虚拟列
CREATE INDEX idx_json_Horizontal_region ON R1(vHorizontal_region);
--在虚拟列上创建索引
```

6. 关系数据输出为JSON数据格式

```
Select * from NATION FOR JSON AUTO;      --将查询结果自动输出为Json格式
Select * from NATION FOR JSON PATH, ROOT('Nations'); --在Json数据中增加名为Nations的根节点
```

SQL for JSON数据管理

使用for json子句，将查询结果作为json字符串导出

select ccolumn, expression, column as alias from table1, table2, table3 for json [auto | path]

- **有两种类型的for json子句：**

（1）**FOR JSON Path**，通过列名或者列别名来定义JSON对象的层次结构，列别名中可以包含“.”，JSON的成员层次结构将会与别名中的层次结构保持一致。

该特性非常类似于早期SQL Server版本中的For Xml Path子句，可以使用斜线来定义xml的层次结构。

（2）**FOR JSON Auto**，自动按照查询语句中使用的表结构来创建嵌套的JSON子数组，类似于For Xml Auto特性。

Input table data:

Number	Date	Customer	Price	Quantity
SO43659	2011-05-31T00:00:00	MSFT	59.99	1
SO43661	2011-06-01T00:00:00	Nokia	24.99	3

Query with FOR JSON clause:

```
SELECT  Number AS [Order.Number], Date AS [Order.Date],
        Customer AS Account,
        Price AS 'Item.UnitPrice', Quantity AS 'Item.Qty'
FROM SalesOrder
FOR JSON PATH, ROOT('Orders')
```

JSON output:

```
{
  "Orders":
    [
      {
        "Order": {
              "Number":"SO43659",
              "Date":"2011-05-31T00:00:00"
            },
        "Account": "Microsoft",
        "Item": {
              "Price":59.99,
              "Quantity":1
            }
      },
      {
        "Order":{
            "Number":"SO43661",
            "Date":"2011-06-01T00:00:00"
        },
        "Account": "Nokia",
        "Item":{
            "Price":24.99,
            "Quantity":3
        }
      }
    ]
}
```

SQL for JSON数据管理

使用for json子句

select column, expression, column as alias from table1, table2, table3 for json [auto | path];

- 应用场景：
  - 把需要返回给客户端的一组对象序列化为JSON。
  - 在一对多的父子表关系场景，若不想创建子表，而是想把子表的记录以JSON数组的格式作为父表的一列。
- 例：父表SalesOrderHeader和子表SalesOrderDetails中，一个订单记录对应多个订单明细记录，可以把每个订单的多个商品详情格式化为JSON数组保存到SalesOrderHeader表中的一列。

MySQL之JSON扩展

MySQL 5.7.8版本之前不能直接操作JSON类型数据，可以将一个字段设定成varchar类型，里面存放JSON格式数据；自5.7.8版本开始支持json结构的数据存储和查询。

- **创建带JSON字段的表**

```
CREATE TABLE muscleape (
    id TINYINT UNSIGNED NOT NULL AUTO_INCREMENT,
    category JSON,
    tags JSON,
    PRIMARY KEY ( id )
);
```

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MYSQL提供的JSON函数

	Name	Description
创建JSON值	JSON_ARRAY()	创建 JSON 数组
	JSON_OBJECT()	创建JSON对象
	JSON_QUOTE()	把JSON文档用引号括起来
修改JSON值	JSON_ARRAY_APPEND()	将数据追加到JSON文档
	JSON_ARRAY_INSERT()	插入JSON数组
	JSON_INSERT()	将数据插入JSON文档
	JSON_MERGE_PATCH()	合并JSON文档，替换重复键的值
	JSON_MERGE_PRESERVE()	合并JSON文档，保留重复键的值
	JSON_REMOVE()	从JSON文档中删除数据
	JSON_REPLACE()	替换JSON文档中的值
	JSON_SET()	插入JSON数据
	JSON_UNQUOTE()	去掉JSON值外面的引号
返回JSON值属性	JSON_DEPTH()	JSON文档的最大深度
	JSON_LENGTH()	JSON文档中的元素个数
	JSON_TYPE()	JSON值的数据类型
	JSON_VALID()	JSON值是否有效

https://dev.mysql.com/doc/refman/8.0/en/json-functions.html

MYSQL提供的JSON函数（续）

	Name	Description
查询JSON值	JSON_CONTAINS()	JSON文档在路径中是否包含特定对象
	JSON_CONTAINS_PATH()	JSON文档中是否包含指定路径
	JSON_EXTRACT()	从JSON文档返回数据
	JSON_KEYS()	JSON文档中的键数组
	JSON_OVERLAPS()	比较两个JSON文档，如果它们有任何共同的键值对或数组元素，则返回1，否则0
	JSON_SEARCH()	返回JSON文档中给定字符串的路径。
	JSON_VALUE()	在提供的路径指向的位置从JSON文档中提取值；将该值作为VARCHAR(512)或指定类型返回
JSON集函数	JSON_ARRAYAGG()	通过聚合 SQL 数据或列来构造 JSON 数组
	JSON_OBJECTAGG()	通过聚合 SQL 数据或列来构造 JSON 对象
JSON表函数	JSON_TABLE()	将JSON表达式中的数据作为关系表返回
JSON工具函数	JSON_PRETTY()	以可读格式打印JSON文档

https://dev.mysql.com/doc/refman/8.0/en/json-functions.html

MySQL之JSON扩展

- 创建JSON值：

1）JSON_ARRAY([val[, val] ...])：创建 JSON 数组。

```
SELECT JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME());
```

```
| JSON_ARRAY(1, "abc", NULL, TRUE, CURTIME()) |
+---------------------------------------------+
| [1, "abc", null, true, "11:30:24.000000"]   |
```

2）JSON_OBJECT([key, val[, key, val] ...])：创建 JSON 对象。

```
SELECT JSON_OBJECT('id', 87, 'name', 'carrot');
```

```
| JSON_OBJECT('id', 87, 'name', 'carrot') |
+-----------------------------------------+
| {"id": 87, "name": "carrot"}            |
```

3）JSON_QUOTE(string)：通过用双引号字符封装字符串，并转义内部引号和其它字符，将其作为 JSON 值，然后将结果作为 utf8mb4 字符串返回。

```
SELECT JSON_QUOTE('[1, 2, 3]');
```

```
| JSON_QUOTE('[1, 2, 3]') |
+-------------------------+
| "[1, 2, 3]"             |
```

MySQL之JSON扩展

- **插入JSON数据**

1）插入 json 格式的字符串，可以是对象的形式，也可以是数组的形式;

例：

```
INSERT INTO muscleape (category, tags)
VALUES ('{"id": 1,"name":
"muscleape"}','[1,2,3]' );
```

or **使用转义符**：

```
INSERT INTO muscleape (category, tags)
VALUES ("{\"id\": 1,\"name\": \"muscleape\"}",
"[1,2,3]");
```

*--在双引号字符前添加一个转义符 *

2）可以使用JSON_OBJECT、JSON_ARRAY函数生成;

例：

```
INSERT INTO muscleape (category, tags)
VALUES
(JSON_OBJECT("id",2,"name","muscleape_q"),
JSON_ARRAY(1,3,5));
```

结果： Select * from muscleape;

id	category	tags
1	{"id": 1, "name": "muscleape"}	[1, 2, 3]
2	{"id": 2, "name": "muscleape_q"}	[1, 3, 5]

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

id	category	tags
1	{"id": 1, "name": "muscleape"}	[1, 2, 3]
2	{"id": 2, "name": "muscleape_q"}	[1, 3, 5]

- 查询JSON

1）用 column->path的形式，其中对象类型path的表示方式 $.path，数组类型的表示方式 $[index]。

例：

```
SELECT id, category->'$.id', category->'$.name', tags->'$[0]',tags->'$[2]' FROM muscleape;
```

id	category->'$.id'	category->'$.name'	tags->'$[0]'	tags->'$[2]'
1	1	"muscleape"	1	3
2	2	"muscleape_q"	1	5

2）查询结果中字符串类型还包含有双引号，可以使用JSON_UNQUOTE函数将双引号去掉，从MySQL 5.7.13开始也可以使用操作符 ->>。

例：

```
SELECT id, category->'$.name', JSON_UNQUOTE(category->'$.name'), tags->'$[0]',tags->'$[2]'  FROM muscleape;
```

id	category->'$.name'	JSON_UNQUOTE(category->'$.name')	category->>'$.name'	id
1	"muscleape"	muscleape	muscleape	1
2	"muscleape_q"	muscleape_q	muscleape_q	2

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

- **查询JSON**

3）JSON_EXTRACT(json_doc, path[, path] ...)，返回 JSON 文档中的数据，该数据是从路径参数匹配的文档部分中选择的。 返回值由path匹配的所有值组成，如果返回多个值，则自动封装为数组，顺序与生成它们的路径相对应。

例1：
```
SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]');
```
结果： 20

例2：
```
SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[1]', '$[0]');
```
结果： [20, 10] *--多个值拼接为数组*

例3：
```
SELECT JSON_EXTRACT('[10, 20, [30, 40]]', '$[2][*]');
```
结果： [30, 40]

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/json-search-functions.html

MySQL之JSON扩展

id	category	tags
1	{"id": 1, "name": "muscleape"}	[1, 2, 3]
2	{"id": 2, "name": "muscleape_q"}	[1, 3, 5]

- JSON 作为条件搜索

1) JSON不同于字符串，如果直接和JSON字段比较，不会查询到结果。

例： SELECT * FROM muscleape WHERE category = '{"id": 1,"name": "muscleape"}';

--结果查询不到数据。

需要使用CAST将字符串转成JSON的形式：

例： SELECT * FROM muscleape WHERE category = CAST('{"id": 1,"name": "muscleape"}' AS JSON);

id	category	tags
1	{"id": 1, "name": "muscleape"}	[1, 2, 3]

2) column->path形式：之前提到过column->path形式从select中查询出的字符串包含双引号，但这里作为查询条件是没有影响的，->和->>的结果是一样的。

例：以下的查询结果相同：

(1) SELECT * FROM muscleape WHERE category->'$.name' = 'muscleape';

(2) SELECT * FROM muscleape WHERE category ->>'$.name' = 'muscleape';

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

id	category	tags
1	{"id": 1, "name": "muscleape"}	[1, 2, 3]
2	{"id": 2, "name": "muscleape_q"}	[1, 3, 5]

- JSON 作为条件搜索

3) JSON_CONTAINS(target, candidate[, path])，其**第二个参数不接受整型**，无论JSON元素是整型还是字符串。如果提供了path，指示是否在目标path中找到该候选 JSON 文档。

例1：

```
SELECT * FROM muscleape
  WHERE JSON_CONTAINS(category,1,'$.id');
```

--执行结果报错：ERROR 3146 (22001): Invalid data type for JSON data in argument 2 to function json_contains; a JSON string or JSON type is required.

例2：

```
SELECT * FROM muscleape
  WHERE JSON_CONTAINS(category,'1','$.id');
```

-- 可以查询到数据，通过返回 1 或 0 指示给定的候选 JSON 文档是否包含在目标 JSON 文档中。

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

- **更新JSON**

1）新整个JSON，与插入时类似。

例：
```
UPDATE muscleape SET tags = '[1, 3, 4]'  WHERE id = 1;
```

2）JSON_INSERT( )函数，插入新值，但不会覆盖已存在的值。

例：
```
UPDATE muscleape SET category = JSON_INSERT(category,
  '$.name','muscleape_new','$.url','muscleape.com')  WHERE id = 1;
```

--当JSON数据中已经存在name属性而没有url属性时，name值不会被修改，而url的值被添加进去。

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

- 更新JSON

3）JSON_SET( )函数，插入新值，并覆盖已存在的值。

例：
```
UPDATE muscleape SET category = JSON_SET(category,
'$.host', 'localhost', '$.url', 'www.muscleape.com') WHERE id = 1;
```

4）JSON_REPLACE( )函数，只替换已存在的值。

例：
```
UPDATE muscleape SET category = JSON_REPLACE(category,
'$.host', '127.0.0.1', '$.address', 'shandong') WHERE id = 1;
```

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

- JSON 更新

5）JSON_ARRAY_APPEND( )函数，在path**指定位置处的数组的尾部增加元素。**

JSON_ARRAY_APPEND(json_doc, path, val[, path, val] ...)

例1： UPDATE muscleape SET tags = JSON_ARRAY_APPEND(tags,'$[0]',4) WHERE id = 1;

id	category	tags
1	{"id": 1, "name": "muscleape"}	[[1, 4], 2, 3]

例2： SET @j = '["a", ["b", "c"], "d"]';
SELECT JSON_ARRAY_APPEND(@j, '$[1][0]', 3);

--指定位置并非数组，则扩充为数组，结果：

["a", [["b", 3], "c"], "d"]

例3： SET @j = '{"a": 1, "b": [2, 3], "c": 4}';
SELECT JSON_ARRAY_APPEND(@j, '$.b', 'x');

结果： {"a": 1, "b": [2, 3, "x"], "c": 4}

例4： SET @j = '{"a": 1}';
SELECT JSON_ARRAY_APPEND(@j, '$', 'z');

结果： [{"a": 1}, "z"]

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/json-modification-functions.html

MySQL之JSON扩展

- JSON 更新

6）JSON_ARRAY_INSERT( )，在path**指定的数组下标位置处**插入元素：

JSON_ARRAY_INSERT(json_doc, path, val[, path, val] ...)

例1：

```
UPDATE muscleape SET tags = JSON_ARRAY_INSERT(tags,'$[0]',5) WHERE id = 1;
```

id	category	tags
1	{"id": 1, "name": "muscleape"}	[5, [1, 4], 2, 3]

例2：

```
SET @j = '["a", {"b": [1, 2]}, [3, 4]]';
SELECT JSON_ARRAY_INSERT(@j, '$[2][1]', 'y');
```

结果： ["a", {"b": [1, 2]}, [3, "y", 4]]

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/json-modification-functions.html

MySQL之JSON扩展

id	category	tags
1	{"id": 1, "name": "muscleape"}	[1, 2, 3]
2	{"id": 2, "name": "muscleape_q"}	[1, 3, 5]

- **JSON 更新**

6）JSON_ARRAY_INSERT( )，在path指定的数组下标位置处插入元素。

例3： `mysql> SET @j = '["a", {"b": [1, 2]}, [3, 4]]';`

`mysql> SELECT JSON_ARRAY_INSERT(@j, '$[1].b[0]', 'x');`

结果： ["a", {"b": ["x", 1, 2]}, [3, 4]]

- **删除JSON元素**

JSON_REMOVE( )函数：

例1： `UPDATE muscleape SET category = JSON_REMOVE(category,'$.host', '$.url')`
`WHERE id = 1;` *--删除属性*

例2： `UPDATE muscleape SET tags = JSON_REMOVE(tags, '$[0]') WHERE id = 1;`

--删除json属性tags的第一个数组元素

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/json-modification-functions.html

MySQL之JSON扩展

- **创建索引**

MySQL的JSON格式数据不能直接创建索引，但是可以**用虚拟列的方式变通，把要搜索的数据单独构建一个虚拟数据列，然后在这个虚拟数据列上创建一个索引。**

例：

```
CREATE TABLE muscleape_office
(    c JSON,
     g INT GENERATED ALWAYS AS (c->"$.id"),
     INDEX i (g)
);

DESC muscleape_office; --查看表结构
```

Field	type	Null	Key	Default	Extra
c	json	YES	""	""	
g	int(11)	YES	MUL		VIRTUAL GENERATED

```
INSERT INTO muscleape_office (c)VALUES
  ('{"id": "1", "name": "Fred"}'),
  ('{"id": "2", "name": "Wilma"}'),
  ('{"id": "3", "name": "Barney"}'),
  ('{"id": "4", "name": "Betty"}');

SELECT c->>'$.name' AS name FROM muscleape_office
WHERE g > 2;
```

name
Barney
Betty

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851

MySQL之JSON扩展

- **MySQL for JSON支持的索引：多值索引 和 虚拟列上的辅助索引。**
- **多值索引**：多值索引是在存储数组值的列上定义的辅助索引。 多值索引也可以定义为复合索引的一部分。

```
CREATE TABLE customers (
    id BIGINT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    modified DATETIME DEFAULT CURRENT_TIMESTAMP,
    custinfo JSON
    );
```

在 WHERE 子句中指定以下函数时，优化器使用多值索引来获取记录：

- MEMBER OF()
- JSON_CONTAINS()
- JSON_OVERLAPS()

```
//建表后修改表：
ALTER TABLE customers ADD INDEX zips( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) );
//建表后建索引：
CREATE INDEX zips ON customers ( (CAST(custinfo->'$.zipcode' AS UNSIGNED ARRAY)) );
```

https://blog.csdn.net/wzy0623/article/details/139503850

MySQL之JSON扩展

- **从JSON中提取关系表**

  JSON_TABLE(expr, path COLUMNS (column_list) [AS] alias)

  从一个指定的JSON文档中提取数据并返回一个具有指定列的关系表，其中：

  *-- **Expr** 表示JSON数据*

  *-- **column_list**: column[, column][, ...]*

  ***column**:*

  *name **FOR ORDINALITY***

  *| name type **PATH** string path [on_empty] [on_error]*

  *| name type **EXISTS PATH** string path*

  *| **NESTED [PATH] path COLUMNS** (column_list)*

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/json-table-functions.html

MySQL之JSON扩展

●从JSON中提取关系表

1）name FOR ORDINALITY，生成一个从 1 开始的计数器列，名字为 name。

例1：将数组中的每个对象元素转为一个关系表中的一行，表中的列对应了每个对象中的成员，其中for ordinality定义了自增长计数器列。

```
例：  SELECT * FROM JSON_TABLE( '[{"x": 10, "y": 11}, {"x": 20, "y": 21}]',
        '$[*]'      --表示数组中的所有元素
          COLUMNS ( id FOR ORDINALITY,
          x INT PATH '$.x',
          y INT PATH '$.y' )  ) AS t;
```

id	x	y
1	10	11
2	20	21

2）提取数组中指定行

例2：提取数组中第2行，设置值为空时的缺省值。

```
SELECT * FROM JSON_TABLE( '[{"x": 10, "y": 11}, {"x": 20}]',
        '$[1]'  --取数组中的第2个元素
          COLUMNS ( id FOR ORDINALITY,
          x INT PATH '$.x',
          y INT PATH '$.y' DEFAULT '100'
                          ON EMPTY )
          ) AS t;
```

id	x	y
2	20	100

参考链接：https://www.aimzhi.cn/doc/mysql/999688f954d4c656.html； https://dev.mysql.com/doc/refman/8.1/en/json-table-functions.html

MySQL之JSON扩展

- 从JSON中提取关系表

3）拉平内嵌的数组

例3：
```
SELECT * FROM
JSON_TABLE(  '[{"x":10,"y":[11, 12]},
{"x":20,"y":[21, 22]}]',
  '$[*]'  COLUMNS (  x INT PATH '$.x',
 NESTED PATH '$.y[*]' COLUMNS (y INT PATH
'$') )
 --展开 y 对应的数组，并将 y 数组中的每个元素
放入名称为 y 的列中
) AS t;
```

x	y
10	11
10	12
20	21
20	22

例4：**Sibling nested paths**，在COLUMNS子句中有多个NESTED PATH实例，转换为数组时分别逐个处理每个NESTED PATH，每次处理时其他兄弟NESTED PATH对应列被赋为空值。

```
SELECT * FROM JSON_TABLE( '[{"a": 1, "b":
[11,111]}, {"a": 2, "b": [22,222]}]',
   '$[*]' COLUMNS(  a INT PATH '$.a',
   NESTED PATH '$.b[*]' COLUMNS (b1 INT PATH
'$'),
   NESTED PATH '$.b[*]' COLUMNS (b2 INT PATH
'$') )
) AS jt;
```

a	b1	b2
1	11	null
1	111	null
1	null	11
1	null	111
2	22	null
2	222	null
2	null	22
2	null	222

参考链接：https://www.aimzhi.cn/doc/mysql/999688f954d4c656.html； https://dev.mysql.com/doc/refman/8.1/en/json-table-functions.html

MySQL之JSON扩展

- 从JSON中提取关系表

4）拉平内嵌的对象

例5：

```
SELECT * FROM JSON_TABLE(
  '[{"x":10,"y":{"a":11,"b":12}},
  {"x":20,"y":{"a":21,"b":22}}]',
   '$[*]' COLUMNS ( x INT PATH '$.x',
     NESTED PATH '$.y'
     COLUMNS ( ya INT PATH '$.a',
               yb INT PATH '$.b' ))
  ) AS t;
```

x	ya	yb
10	11	12
20	21	22

参考链接：https://www.aimzhi.cn/doc/mysql/999688f954d4c656.html； https://dev.mysql.com/doc/refman/8.1/en/json-table-functions.html

MySQL之JSON扩展

c1	c2	at	bt	ct	JSON_EXTRACT(c3, '$.*')
2	x	1	2	3	[1, 15]
2	x	1	2	3	[1, 15]
3	w	1	2	3	[5, 6, 7]
3	w	1	2	3	[5, 6, 7]
3	w	1	2	3	[5, 6, 7]
5	v	1	2	3	[123, 1111]
5	v	1	2	3	[123, 1111]

- JSON_TABLE的**延迟导出表**(lateral derived table) 连接

MySQL不支持在FROM子句中对**依赖于前面的关系的导出表**和这些前面的关系进行join操作，但对于**table函数**是例外的，这些table函数生成的表被视作延迟导出表。

例1：

```
CREATE TABLE t1 (c1 INT, c2 CHAR(1), c3 JSON);

INSERT INTO t1 ( ) VALUES
  ROW(1, 'z', JSON_OBJECT('a', 23, 'b', 27, 'c', 1)),
  ROW(1, 'y', JSON_OBJECT('a', 44, 'b', 22, 'c', 11)),
  ROW(2, 'x', JSON_OBJECT('b', 1, 'c', 15)),
  ROW(3, 'w', JSON_OBJECT('a', 5, 'b', 6, 'c', 7)),
  ROW(5, 'v', JSON_OBJECT('a', 123, 'c', 1111)) ;
```

```
SELECT c1, c2, t.at, tt.bt, tt.ct, JSON_EXTRACT(c3, '$.*')
FROM t1 AS m
JOIN
JSON_TABLE(   m.c3,
  '$.*'  COLUMNS(
    at VARCHAR(10) PATH '$.a' DEFAULT '1' ON EMPTY,
    bt VARCHAR(10) PATH '$.b' DEFAULT '2' ON EMPTY,
    ct VARCHAR(10) PATH '$.c' DEFAULT '3' ON EMPTY
  )
) AS tt
ON m.c1 > tt.at;
```

如果要分析大规模的JSON数据，JSON_TABLE比单个的JSON属性值函数更适合，它将这些数据转换成对应的MySQL表。

参考链接：https://dev.mysql.com/doc/refman/8.1/en/json-table-functions.html

MySQL之JSON扩展

- JSON_TABLE的**延迟导出表**(lateral derived table) 连接

对比SQLSERVER for JSON的CROSSAPPLY功能?

例2:

```
CREATE TABLE t4 (c1 INT, c2 CHAR(1), c3 JSON);
INSERT INTO t4 ( ) VALUES (1, 2, '{"name":
   [{"a":1,"b":"action"}, {"a":1,"b":"shard"},{"a":2,"b":"oracle"}]}');

SELECT c1, c2 , tt.at, tt.bt, tt.ct, JSON_EXTRACT(c3, '$.*')
FROM t4 AS m, JSON_TABLE( m.c3,   '$.name[*]' COLUMNS(
      at VARCHAR(10) PATH '$.a' DEFAULT '1' ON EMPTY,
      bt VARCHAR(10) PATH '$.b' DEFAULT '2' ON EMPTY,
      ct VARCHAR(10) PATH '$.c' DEFAULT '3' ON EMPTY  )) AS tt WHERE m.c1 = tt.at;
```

c1	c2	at	bt	ct	JSON_EXTRACT(c3, '$.*')
1	2	1	action	3	[[{"a": 1, "b": "action"}, {"a": 1, "b": "shard"}, {"a": 2, "b": "oracle"}]]
1	2	1	shard	3	[[{"a": 1, "b": "action"}, {"a": 1, "b": "shard"}, {"a": 2, "b": "oracle"}]]

参考链接：https://dev.mysql.com/doc/refman/8.1/en/json-table-functions.html

MySQL之JSON扩展

Over clause与patition动作关联

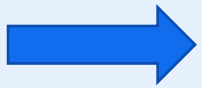
- Json聚合

1）JSON_ARRAYAGG(col_or_expr) [over_clause]，将某列或者表达式值汇总为JSON数组。

例：

```
SELECT o_id, JSON_ARRAYAGG(attribute) AS attributes
FROM t3 GROUP BY o_id;
```

o_id	attribute	val
2	color	red
2	fabric	silk
3	color	green
3	shape	square



o_id	attributes
2	["color", "fabric"]
3	["color", "shape"]

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/aggregate-functions.html#function_json-arrayagg

MySQL之JSON扩展

- Json聚合

2）JSON_OBJECTAGG(key, value) [over_clause]，将某列或者表达式值汇总为JSON对象。

例：

```
SELECT o_id, JSON_OBJECTAGG(attribute, val)
FROM t3 GROUP BY o_id;
```

o_id	attribute	val
2	color	red
2	fabric	silk
3	color	green
3	shape	square

o_id	attributes
2	{"color": "red", "fabric": "silk"}
3	{"color": "green", "shape": "square"}

参考链接：https://blog.csdn.net/qq_16946803/article/details/127620851；https://dev.mysql.com/doc/refman/8.1/en/aggregate-functions.html#function_json-arrayagg

2.2 关系数据库标准语言SQL
2.2.2 面向大数据管理的SQL扩展语法（续）

- **SQL与R语言集成：** *R作为一种统计分析软件，集统计分析与图形显示于一体*
- **SQL-R连接与SQL语句执行**
- **数据库读入——RODBC包**

RODBC包提供了ODBC的访问接口，调用数据库中的数据

功能	作用
odbcConnect	打开一个连接，返回一个数据库访问句柄
sqlSave	把R数据框复制到一个数据库的表中
sqlFetch	把一个数据库中的表拷贝到一个R数据框中
sqlQuery	查询，返回的结果是R的数据框
odbcClose	关闭连接

R语言环境中使用SQL搜索sqldf包

- 组合使用：RODBC从数据库读入环境，sqldf进行搜索。

1. 数据筛选与排序

```
sqldf("select * from sale where market='东' order by year")
```

2. 数据合并——纵向连接 union，except，intersect

```
UNION_all<-sqldf("select * from one union all select * from two")
```

3. 数据合并——横向连接 内连接，左连接

```
inner1<-sqldf("select * from table1 as a inner join table2 as b on a.id=b.id"); #内连接
inner2<-sqldf("select * from table1 as a, table2 as b where a.id=b.id"); #笛卡尔积
left3<-sqldf("select * from table1 as a left join table2 as b on a.id=b.id"); #左连接
```

阅读教材P30-33决策树的例子

2.3 SQL on Hadoop

通过Hadoop大数据平台扩展SQL的分布式查询处理能力，可分为四种类型：

（1）**SQL outside Hadoop**

通过连接器，实现SQL语言直接访问Hadoop数据。如：Vertica、Teradata、Oracle等。

（2）**SQL alongside Hadoop**

Hadoop与数据库的混合架构，通过修改的SQL引擎将负载分布在SQL和MapReduce引擎。

（3）**SQL on Hadoop**

在Hadoop系统中集成SQL功能，一种是提供类SQL功能，实则转换为MapReduce动作执行，另一种是参照关系数据库的MPP架构不使用MapReduce，而是在HDFS上实现执行计划树并分派到各个节点执行。如：Impala、Presto等。

（4）**SQL in Hadoop**

将关系数据库成熟技术与Hadoop紧密结合，实现Hadoop中的数据库。如：Actian VetorH。

2.3 SQL on Hadoop

- **SQL on Hadoop：HiveQL**

Apache Hive上的一种类SQL语言，通过类似SQL的语法为用户提供Hadoop上数据管理与查询处理能力。

HiveQL的基本语法格式：

```
SELECT [ALL | DISTINCT] select_expr, select_expr, ...
FROM table_reference
[WHERE where_condition]
[GROUP BY col_list]
[HAVING having_condition]
[CLUSTER BY col_list | [DISTRIBUTE BY col_list] [SORT BY| ORDER BY col_list]]
[LIMIT number];
```

DISTRIBUTE BY: 分区排序，类似MapReduce中partition，进行分区，结合sort by使用。

SORT BY：每个MapReduce内部进行排序，不是对全局结果集排序，只在本机做排序。

ORDER BY：对应全局排序，只有一个reduce任务。

CLUSTER BY：当distribute by和sort by字段相同时，可以换成使用cluster by方式。cluster by除了具有distribute by的功能外还兼具sort by的功能。

SQL on Hadoop：HiveQL

- HiveQL连接示例（不直接支持等值连接条件）：

```
select l_orderkey, sum(l_extendedprice*(1-l_discount)) as revenue, o_orderdate,
o_shippriority
from customer c join orders o on c.c_mktsegment='building'
and c.c_custkey=o.o_orderkey
where o_orderdate<'1995-03-15'and l_shipdate>'1995-03-15'
group by l_orderkey,o_orderdate,o_shippriority
order by revenue desc,o_orderdate;
```

SQL on Hadoop：HiveQL

- HiveQL除了支持常见的SQL数据结构，还支持数组、结构体、映射数据类型，在查询功能上，还支持嵌入MapReduce程序。
- 例：HiveQL调用MapReduce程序

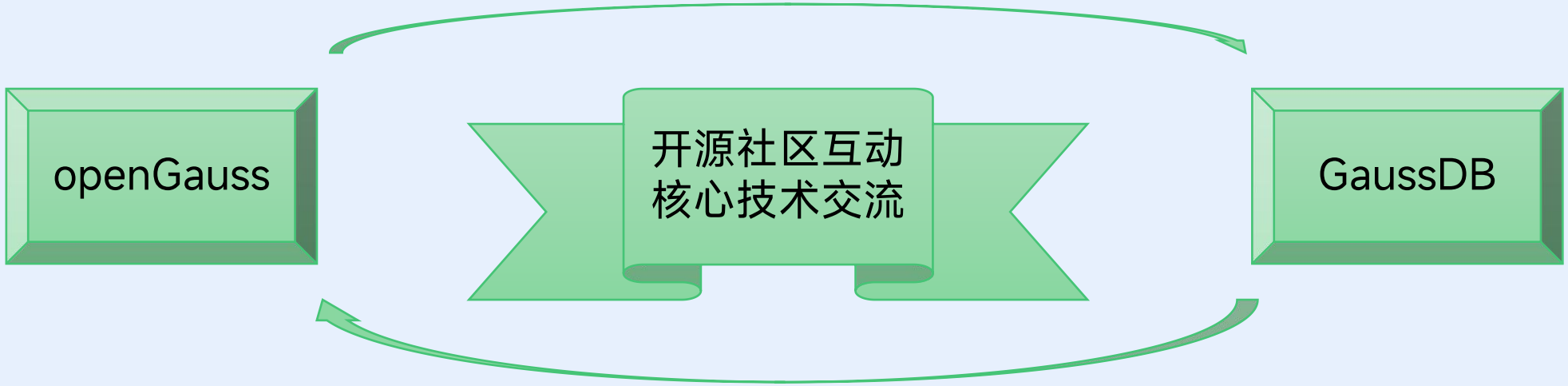
```
FROM (
FROM docs
MAP doctext
USING ‘python wordcount_mapper.py’as (word, cnt)
CLUSTER BY word) it
REDUCE it.word, it.cnt USING ‘python wordcount_reduce.py’;
```

Doctext是输入，word、cnt是map输出，cluster by对word哈希分区后作为Reduce的输入

（拓展*）国产数据库之JSON扩展-openGauss

OpenGauss是华为数据库的开源生态下的研发成果，基于此，衍生出了企业级分布式关系型数据库GaussDB系列数据库产品，主要面对云数据库领域。

这样的技术交流和产品研发方式，体现出了国内IT产品研发领域的一种开放式氛围和良性互动。通过开源社区分项核心技术，孕育技术人才，通过社区技术交流推动技术创新衍生新型产品。



（拓展*）国产数据库之JSON扩展-OpenGauss

对于JSON数据的三种基本形态：标量、数组和键值对象，

OpenGauss将其中的数组和对象统称容器（container），并在系统内提供两种数据类型**JSON**和**JSONB**支持JSON数据存储。

- **JSON**，是对输入的字符串的完整拷贝，使用时再去解析，会保留输入的空格，重复键以及顺序等；
- **JSONB**，是解析输入后保存的二进制数据，它在解析时会删除语义无关的细节和重复的键，对键值也会进行排序，使用时无需再次解析。
  - 输入相对JSON格式慢，但处理更快；
  - JSONB支持索引
  - JSONB类型存在解析后的格式归一化等操作，同等的语义下只会有一种格式，因此可以更好的支持很多其他额外的操作，例如按照一定的规则进行大小比较等。

（拓展*）国产数据库之JSON扩展-OpenGauss的JSON输入格式

- **输入格式**：必须是一个符合JSON数据格式的字符串，用单引号’‘ 声明。

（1）输入空值null：仅null，全小写。

例：

```
select 'null'::json;
```

错误：

```
select 'NULL'::jsonb;
```

（2）输入数字：正负整数、小数、0，支持科学计数法。

例：

```
select '1'::json;  select '-1.5'::json;
select '-1.5e-5'::jsonb, '-1.5e+2'::jsonb;
```

错误：

```
select '001'::json, '+15'::json, 'NaN'::json;
-- 不支持多余的前导0，正数的+号，以及NaN和infinity。
```

（拓展*）国产数据库之JSON扩展-OpenGauss的JSON输入格式

- **输入格式：**

（3）布尔（bool-json）：true、false，全小写。

例：select 'true'::json;　　select 'false'::jsonb;

（4）字符串（str-json）：必须是加双引号的字符串。

例：select '"a"'::json;　　select '"abc"'::jsonb;

（5）数组（array-json）：使用中括号[]包裹，满足数组书写条件。数组内元素类型可以是任意合法的JSON，且不要求类型一致。

例：select '[1, 2, "foo", null]'::json;　　select '[ ]'::json;

　　select '[1, 2, "foo", null, [[]], {}]'::jsonb;

（拓展*）国产数据库之JSON扩展-OpenGauss的JSON输入格式

- **输入格式：**

（6）对象（object-json）：使用大括号{}包裹，键必须是满足JSON字符串规则的字符串，值可以是任意合法的JSON。

例： select '{}'::json;

select '{"a": 1, "b": {"a": 2, "b": null}}'::json;

select '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::jsonb;

注意：

区分 **'null'::json** 和 **null::json** 是两个不同的概念，类似于字符串 str="" 和 str=null。

对于数字，当使用科学计数法的时候，jsonb类型会将其展开，而json会精准拷贝输入。

（拓展*）国产数据库之JSON扩展-OpenGauss的JSONB优化机制

JSONB的优化机制

JSONB存储的是解析后的二进制，能够体现JSON的层次结构，更方便直接访问等，因此JSONB有优于JSON的特性。

（1）格式归一化

（1.1）输入的json字符串解析成jsonb二进制后，会丢弃语义上无关的细节，例如空格。

例：openGauss=# select ' [1, " a ", {"a" :1 }] '::jsonb;

[1, " a ", {"a": 1}](1 row)

（拓展*）国产数据库之JSON扩展-OpenGauss的JSONB优化机制

（1.2）删除重复的键值，只保留最后一个出现的。

例： openGauss=# select '{"a" : 1, "a" : 2}'::jsonb;

```
----------
{"a": 2}(1 row)
```

（1.3）键值会重新进行排序，排序规则：长度长的在后、长度相等则ascii码大的在后。

例： openGauss=# select '{"aa" : 1, "b" : 2, "a" : 3}'::jsonb;

```
--------------------------
{"a": 3, "b": 2, "aa": 1}(1 row)
```

（拓展*）国产数据库之JSON扩展-OpenGauss的JSONB优化机制

（2）大小比较

（2.1）首先比较类型

object-jsonb > array-jsonb > bool-jsonb > num-jsonb > str-jsonb > null-jsonb

（2.2）同类型则比较内容

- str-json类型：依据text比较的方法，使用数据库默认排序规则进行比较，返回值正数代表大于，负数代表小于，0表示相等。
- num-json类型：数值比较
- bool-json类型：true > false
- array-jsonb类型：长度长的 > 长度短的，长度相等则依次比较每个元素。
- object-jsonb类型：长度长的 > 长度短的，长度相等则依次比较每个键值对，先比较键，在比较值。

（拓展*）国产数据库之JSON扩展-OpenGauss的JSONB优化机制

（3）创建索引、主外键

（3.1）BTREE索引

JSONB类型支持创建btree索引，支持创建主键、外键。

（3.2）GIN索引

GIN索引（通用倒排索引）可以用来有效的搜索出现在大量JSONB文档（datums）中的键或者键/值对。

（3.3）包含存在

查询一个JSON之中是否包含某些元素，或者某些元素是否存在于某个JSON中是JSONB的一个重要能力。

```
-- 简单的标量/原始值只包含相同的值。
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
-- 左侧数组包含了右侧字符串。
SELECT '[1, "aa", 3]'::jsonb ? 'aa';
-- 左侧数组包含了右侧的数组所有元素，顺序、重复不重要。
SELECT '[1, 2, 3]'::jsonb @> '[1, 3, 1]'::jsonb;
-- 左侧object-json包含了右侧object-json的所有键值对。
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb":true}'::jsonb
@> '{"version":9.4}'::jsonb;
-- 左侧数组并没有包含右侧的数组所有元素，因为左侧数组的三个元素为1、2、[1,3]，右侧的为1、3。
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;  --false
-- 同上，没有存在包含关系，返回值为false。
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb; -- false
```