# 背景

数据操纵是数据库管理系统中一种最基本的操作，这种操作包括查询、插入、删除和修改等，其中，查询操作称为查询处理。

查询的执行，就是查询处理的过程，即数据库按用户指定的SQL语句中的语义，执行语义所限定的操作。但SQL语句的执行效率对数据库的效率影响较大。为了提高查询语句的执行效率，对查询语句进行优化是必不可少的。

对查询语句进行优化的技术就是查询优化技术，运用查询技术实现数据操纵功能的过程是确定给定查询的高效执行计划的过程。所谓执行计划就是查询树，它由一系列内部的操作符组成，这些操作符按一定的运算关系构成查询的一个**执行方案**。

# 概述

查询优化的追求目标，就是在数据库查询优化引擎生成一个执行策略的过程中，尽量使查询的总开销（总开销通常包括IO、CPU、网络传输等）达到最小。数据库查询优化技术主要包括查询重用技术、查询重写规则、查询算法优化技术、并行查询优化技术、分布式查询优化技术及其他方面（如框架结构）的优化技术，这6项技术构成了一个“广义的数据库查询优化”的概念。

从优化的内容角度看，查询优化又分为**代数优化**和**非代数优化**，或称为**逻辑优化**和**物理优化**。**逻辑优化主要依据关系代数的等价变换做一些逻辑变换，物理优化主要根据数据读取、表连接方式、表连接顺序、排序等技术对查询进行优化**。“查询重写规则”属于逻辑优化方式，运用了关系代数和启发式规则；“查询算法优化”属于物理优化方式，运用了基于代价估算的多表连接算法求解最小花费的技术。

## 代价模型

总代价模型：COST = CPU Cost + IO Cost  
 MySQL在cost类型上分为IO、CPU和Memory，MySQL5.7的代价模型还在完善中，Memory的代价虽然已经收集了，但还没有计算在最终的代价中。  
 MySQL5.7在源码上对cost模型进行了大量重构，代价分为server层和engine层。server层主要是CPU代价，而engine层主要是IO代价。MySQL5.7 引入了两个系统表mysql.server\_cost和mysql.engine\_cost来分别配置这两个层的代价。  
以下分析均基于MySQL5.7.10

### server\_cost

1、row\_evaluate\_cost (default 0.2) 计算符合条件的行的代价，行数越多，此项代价越大；  
 2、memory\_temptable\_create\_cost (default 2.0) 内存临时表的创建代价；  
 3、memory\_temptable\_row\_cost (default 0.2) 内存临时表的行代价；  
 4、key\_compare\_cost (default 0.1) 键比较的代价，例如排序；  
 5、disk\_temptable\_create\_cost (default 40.0) 内部myisam或innodb临时表的创建代价；  
 6. disk\_temptable\_row\_cost (default 1.0) 内部myisam或innodb临时表的行代价；  
 可以看出创建临时表的代价是很高的，尤其是内部的myisam或innodb临时表。

### engine\_cost

1. io\_block\_read\_cost (default 1.0) 从磁盘读数据的cost，对innodb来说，表示从磁盘读一个page的cost；  
    2、memory\_block\_read\_cost (default 1.0）；  
    从内存读数据的cost，对innodb来说，表示从buffer pool读一个page的cost。  
   目前io\_block\_read\_cost和memory\_block\_read\_cost默认值均为1，实际生产中建议酌情调大memory\_block\_read\_cost，特别是对普通硬盘的场景。

注：分布式数据库需要增加网络开销。

MySQL8.0对于代价模型有一定的改进。

## 数据库调优

### 方式

在数据库层面进行调优，有很多的资源、数据库配置参数需要考虑。数据库调优的方式通常有如下几种：

❏人工调优。主要依赖于人，效率低下；要求操作者完全理解常识所依赖的原理，还需要对应用、数据库管理系统、操作系统以及硬件有广泛而深刻的理解。

❏基于案例的调优。总结典型应用案例情况中数据库参数的推荐配置值、数据逻辑层设计等情况，从而为用户的调优工作提供一定的参考和借鉴。但这种方式忽略了系统的动态性和不同系统间存在的差异。

❏自调优。为数据库系统建立一个模型，根据“影响数据库系统性能效率的因素”，数据库系统自动进行参数的配置。

### 阶段

通常，数据库调优主要分为5阶段，如表1-1所示。



涉及的技术主要有以下几种：

❏应用情况的估算。应用的使用方式（把业务逻辑转换为数据库的读写分布逻辑，以读多写少或读写均衡等来区分OLAP和OLTP；应用对数据库的并发情况、并发是否可以池化等）、数据量、对数据库的压力、峰值压力等做一个预估。

❏系统选型策略。确定什么样的数据库可以适用应用需求，并确定使用开源的数据库还是商业的数据库，使用集群还是单机的系统，同时对操作系统、中间件、硬件、网络等进行选型。

❏数据模型的设计。主要根据业务逻辑，从几个角度考虑表的逻辑结构，内容如下。

○E-R模型设计：遵循E-R模型设计原理。偶尔的、适当程度的非规范化可以改善系统查询性能。

○数据逻辑分布策略：目的是减少数据请求中不必要的数据量，只返回用户需要的数据。可用的技术如分区、用E-R模型分表等（如互联网企业典型的用法，根据业务的不同，进行分库、分表等操作）。

○数据物理存储策略：目的是减少IO操作，如启用压缩技术、把索引和表数据的存储分开，不同的表数据分布在不同的表空间上，不同的表空间分布在不同的物理存储上（尤其是读写量大的表空间分布在不同的物理存储上），日志、索引和数据分布在不同的物理存储上等。

○索引：在查询频繁的对象上建立恰当的索引，使索引的正效应大于负效应（索引的维护存在消耗）。

❏SQL设计。编写正确的、查询效率高的SQL语句，依据的主要是“查询重写规则”。编写语句的过程中要注意，要有意识地保障SQL能利用到索引。

❏数据库功能的启用。数据库为提高性能提供了一些功能，可合理使用，具体如下。

○查询重用：根据实际情况进行配置，可缓存查询执行计划、查询结果等。

○数据库参数的设置：可设置合适的参数，如数据缓冲区等。

❏模型系统预运行。在备用系统上模拟实际运行环境，加大压力进行系统测试，提前发现问题。

❏系统监控与分析。在工业环境下，加强对系统的运行监控和日常的分析工作，具体如下。

○应用系统表现：收集用户对应用系统的使用意见、系统存在问题等，因为这些可能是用户在第一时间发现的。

○OS环境监控：实时监控CPU、内存、IO等，并对比实时情况与历史正常情况。

○数据库内部状况监控：一些数据库提供系统表、视图、工具等手段，向用户提供数据库运行过程中内部状况的信息，如锁的情况，这些都需要实时监控，并对比实时情况与历史正常情况。

○日志分析：在数据库的日志、操作系统的日志中找出异常事件，定位问题。

## 查询优化技术

查询优化技术是SQL层面的优化，属于局部优化，有别于“数据库调优”式的全局优化。

### 查询重用技术

查询重用是指尽可能利用先前的执行结果，以达到节约查询计算全过程的时间并减少资源消耗的目的。

目前查询重用技术主要集中在两个方面：

❏查询结果的重用。在缓存区中分配一块缓冲块，存放该SQL语句文本和最后的结果集，当遇到同样的SQL输入时，可直接把结果返回。查询结果的重用技术节约了查询计划生成时间和查询执行过程的时间，减少了查询执行全过程的资源消耗。

❏查询计划的重用。缓存一条查询语句的执行计划及其相应语法树结构。查询计划的重用技术减少了查询计划生成的时间和资源消耗。

查询重用技术有利有弊：弊端，如结果集很大会消耗很大的内存资源，同样的SQL不同用户获取的结果集可能不完全相同；益处，节约了CPU和IO消耗。在使用的过程中，趋利避害，应根据实际情况选用。

注：在分布式数据库实践中，可以根据具体的执行计划复杂程度以及结果集的大小确定是否需要缓存该执行计划和结果集。

### 查询重写规则技术

查询重写是查询语句的一种**等价转换**，即对于任何相关模式的任意状态都会产生相同的结果（相同的关系替代两个表达式中相应的关系，所得到的结果是相同的）。查询重写有两个目标：

❏将查询转换为等价的、效率更高的形式，例如将效率低的谓词转换为效率高的谓词、消除重复条件等。

❏尽量将查询重写为等价、简单且不受表顺序限制的形式，为物理查询优化阶段提供更多的选择，如视图的重写、子查询的合并转换等。

查询重写的依据，是关系代数。关系代数的等价变换规则对查询重写提供了理论上的支持。查询重写后，查询优化器可能生成多个连接路径，可以从候选者中择优。

对查询优化技术进行分类，可有以下4个角度：

❏语法级。查询语言层的优化，基于语法进行优化。

❏代数级。查询使用形式逻辑进行优化，运用关系代数的原理进行优化。

❏语义级。根据完整性约束，对查询语句进行语义理解，推知一些可优化的操作。

❏物理级。物理优化技术，基于代价估算模型，比较得出各种执行方式中代价最小的。

查询重写是基于语法级、代数级、语义级的优化，可以统一归属到逻辑优化的范畴：基于代价估算模型是物理层面的优化，是从连接路径中选择代价最小的路径的过程。

查询重写技术优化思路主要包括：

❏将过程性查询转换为描述性的查询，如视图重写。

❏将复杂的查询（如嵌套子查询、外连接、嵌套连接）尽可能转换为多表连接查询。

❏将效率低的谓词转换为等价的效率高的谓词（如等价谓词重写）。

❏利用等式和不等式的性质，简化WHERE、HAVING和ON条件。

如何改进现有查询重写规则的效率，如何发现更多更有效的重写规则，是查询优化的研究内容之一。常见的查询重写技术类型，每一类都有自己的规则，这些规则没有确定的、统一的规律，但**重写的核心一定是“等价转换”，只有等价才能转换，这是需要特别强调的**。

### 查询算法优化技术

查询优化即求解给定查询语句的高效执行计划（有的书籍称为执行方案）的过程。

查询计划，也称为查询树，它由一系列内部的操作符组成，这些操作符按一定的运算关系构成查询的一个执行方案。简单说，就是先将表A和表B连接得到中间结果，然后再和另外的表C连接得到新的中间方式，直至所有表连接完毕（连接操作就是操作符，这个示例有两个连接操作符。A连接B连接C、C连接B连接A就是两种不同的执行方案，是两个不同的执行计划，查询优化要选出最高效的一个执行方案）。

查询计划，从形式上看是一颗二叉树，树叶是每个单表对象，两个树叶的父结点是一个连接操作符（如左外连接操作符，A left-out join B）连接后的中间结果（另外还有一些其他结点如排序操作等也可以作为中间结果），这个结果是一个临时“关系”，这样直至根结点。

所以从一个查询计划看，涉及的主要“关系结点”包括：

❏单表结点。考虑单表的数据获取方式，是直接通过IO获得数据，还是通过索引获取数据（聚簇索引），或者是通过索引定位数据的位置后再经过IO到数据块中获取数据（非聚簇索引）。这是一个从物理存储到内存解析成逻辑字段的过程，即符合冯·诺依曼体系结构的要求（外存数据读入内存才能被处理）。

❏两表结点。考虑两表以何种方式连接、代价有多大、连接路径有哪些等。表示的是内存中的元组怎么进行元组间的连接。此时，元组通常已经存在于内存中，直接使用即可。这是一个完成用户语义的逻辑操作，但是只是局部操作，只涉及两个具体的关系。完成用户全部语义（用户连接的语义），需要配合多表的连接顺序的操作。不同的连接算法导致的连接效率不同，如数据多时可使用Hash连接，外表数据量小且内表数据量大时可使用嵌套连接，数据如果有序可使用归并连接或先排序后使用归并连接等。

❏多表中间结点。考虑多表连接顺序如何构成代价最少的“执行计划”。决定是AB先连接还是BC先连接，这是一个比较花费大小的运算。如果判断的连接方式太多，也会导致效率问题。多个关系采用不同次序进行连接，花费的CPU资源、内存资源差异可能较大。许多数据库采用左深树、右深树、紧密树3种方式或其中一部分对多表进行连接，得到多种连接路径。

查询优化目的就是生成最好的查询计划。生成最好的查询计划的策略通常有两个：

❏基于规则优化。根据经验或一些已经探知或被证明有效的方式，定义为“规则”（如根据关系代数得知的规则、根据经验得知的规则等），用这些规则化简查询计划生成过程中符合可被化简的操作，使用启发式规则排除一些明显不好的存取路径，这就是基于规则的优化。

❏基于代价优化。根据一个代价评估模型，在生成查询计划的过程中，计算每条存取路径（存取路径主要包括上述3个“关系结点”）的花费，然后选择代价最小的作为子路径，这样直至所有表连接完毕得到一个完整的路径。主流数据库都采用了基于代价策略进行优化的技术。

基于规则优化具有操作简单且能快速确定连接方式的优点，但这种方法只是排除了一部分不好的可能，所以得到的结果未必是最好的；基于代价优化是对各种可能的情况进行量化比较，从而可以得到花费最小的情况，但如果组合情况比较多则花费的判断时间就会很多；查询优化器的实现，多是两种优化策略组合使用，如MySQL和PostgreSQL就采取了基于规则和代价估算的查询优化策略。

多表连接的优化算法中，使用最广泛的算法有如下几种：

❏SYSTEM-R算法。近乎穷举的搜索算法（一种空间搜索算法，其变形算法与其本质相同）。

❏启发式搜索算法。基于规则（基于“启发式规则”抛弃不好的存取路径挑选好的）。

❏贪婪算法。根据某种优化方式，以当前情况为基础做出最优选择，认为每次搜索过的局部存取路径是最优的，然后继续探索与其他表的连接路径。

❏动态规划算法。将待求解的问题分解为若干个子问题（阶段），按顺序求解子问题，前一子问题的解，为后一子问题的求解提供了有用的信息。在求解任一子问题时，列出各种可能的局部解，通过决策保留那些可能达到最优的局部解，丢弃其他局部解。依次解决各子问题，最后一个子问题就是初始问题的解。

❏遗传算法。一种启发式的优化算法，抛弃了传统的搜索方式，模拟自然界生物进化过程，基于自然群体遗传演化机制，采用人工进化的方式对目标空间进行随机化搜索。

我们知道，查询的基本操作是选择、投影和连接。选择和投影的优化规则适用于SPJ（Select-Project-Join）和非SPJ（SPJ+GROUPBY等操作）；连接包括两表连接和多表连接，多表连接是其中最难的，因为多个表连接时可以有多种不同的连接次序，所以查询的执行计划的数目会随着该查询包含的表个数呈指数级增长（最大组合次数是n个关系全排列），当表个数很多时，将导致搜索空间极度膨胀，仅搜索花费最小的查询计划就需要耗费巨大的时间和资源，这是查询优化器实现时需要考虑的问题。

### 并行查询的优化技术

传统单机数据库系统中，给定一个查询（Query），查询优化算法只需找到查询的一个具有最小执行花费的执行计划，这样的计划必定具有最快的响应时间。

在并行数据库系统中，查询优化的目标是寻找具有最小响应时间的查询执行计划，这需要把查询工作分解为一些可以并行运行的子工作。一些商业数据库提供了并行查询的功能，用以优化查询执行操作。

一个查询能否并行执行，取决于以下因素：

❏系统中的可用资源（如内存、高速缓存中的数据量等）。

❏CPU的数目。

❏运算中的特定代数运算符。如A、B、C、D4个表进行连接，每个表的单表扫描可以并行进行；在生成4个表连接的查询计划过程中，可选择A和B连接的同时C和D进行连接，这样连接操作能并行运行。不同商业数据库，对查询并行的实现也不尽相同。在同一个SQL内，查询并行可以分为以下两种：

○操作内并行。将同一操作如单表扫描操作、两表连接操作、排序操作等分解成多个独立的子操作，由不同的CPU同时执行。

○操作间并行。一条SQL查询语句可以分解成多个子操作，由多个CPU执行。

注：分布式数据库实践中，对于多表UNION操作可以采用拆分多个SQL，并行下发执行的优化方式提高效率。

### 分布式查询优化技术

在分布式数据库系统中，查询策略优化（主要是数据传输策略，A、B两结点的数据进行连接，是A结点数据传输到B结点或从B到A或先各自进行过滤然后再传输等）和局部处理优化（传统的单结点数据库的查询优化技术）是查询优化的重点。

注：在分布式数据库中不同节点之间数据传输会耗费大量的网络IO，这个是重点考虑优化的地方。

在查询优化策略中，数据的通信开销是优化算法考虑的主要因素。分布式查询优化以**减少传输的次数和数据量**作为查询优化的目标。所以，分布式数据库系统中的代价估算模型，除了考虑CPU代价和IO代价外，还要考虑通过网络在结点间传输数据的代价。这是分布式并行查询优化技术与传统单结点数据库系统最大的不同之处。

在分布式数据库系统中，代价估算模型如下：

总代价=IO代价+CPU代价+通信代价

### 其他优化技术

数据库的查询性能，还取决于其他一些因素，如数据库集群系统中的SD（Share Disk）集群和SN（Share Nothing）集群，不同的架构查询优化技术也不同。SD集群采用的是共享存储方式，在数据的读写时可能产生读写冲突，所以单表扫描会受到影响；SN集群采用的是非共享式存储方式，所以在考虑了通信代价后单结点的优化方式依然适用。

# 逻辑查询优化

查询优化器在逻辑优化阶段主要解决的问题是：如何找出SQL语句等价的变换形式，使得SQL执行更高效。

一条SQL查询语句结构复杂，包含多种类型的子句，优化操作依赖于表的一些属性信息（如索引和约束等）。可用于优化的思路包括：

❏子句局部优化。每种类型子句都可能存在优化方式，这是子句局部的优化，如等价谓词重写、WHERE和HAVING条件化简中的大部分情况，都属于这种子句范围内的局部优化。

❏子句间关联优化。子句与子句之间关联的语义存在优化的可能，如外连接消除、连接消除、子查询优化、视图重写等都属于子句间的关联优化，因为它们的优化都需要借助其他子句、表定义或列属性等信息进行。

❏局部与整体的优化。需要协同考虑局部表达式和整体的关系，如OR重写并集规则需要考虑UNION操作（UNION是变换后的整体的形式）的花费和OR操作（OR是局部表达式）的花费。

❏形式变化优化。多个子句存在嵌套，可以通过形式的变化完成优化，如嵌套连接消除。

❏语义优化。根据完整性约束、SQL表达的含义等信息对语句进行语义优化。

❏其他优化。根据一些规则对非SPJ做的其他优化、根据硬件环境进行的并行查询优化等。各种逻辑优化技术依据关系代数和启发式规则进行。

## 关系代数

### 基本概念

关系模型的数据结构就是我们在关系数据库中提到的二维结构，是一个横纵结合的二维表。在关系模型中，现实世界的实体以及实体间的各种联系均用关系来表示。

关系是一种对象。

关系的另外一个常用词是表。关系和表基本表达同一含义，只是关系更偏向于理论，表更偏向于实际数据库中可进行增、删、改、查等操作的表对象。

关系的元数据，即表的结构，通常称为列或属性。数据库实现中，有的用field表示，有的用Item表示。

关系的数据，即表的行数据，通常称为元组（tuple），也称为记录（record）。一个表可有多行元组。

对关系进行的操作就是关系运算。关系运算是将一定的运算符作用于一定的关系对象上，得到预期的运算结果（预期就是用户语义，用户语义通过运算表达基本语义，通过对不同对象上的各种运算进行组合表达其对关系操作的真实语义）。运算对象、运算符、运算结果是运算的三大要素，所以关系运算就是关系运算符作用在关系上、得到的结果形式也是关系形式的操作。

关系代数的运算符包含以下4类：

**传统集合运算符**。并（UNION）、交（INTERSECTION）、差（DIFFERENCE）、积（EXTENDED CARTESIAN PRODUCT）。

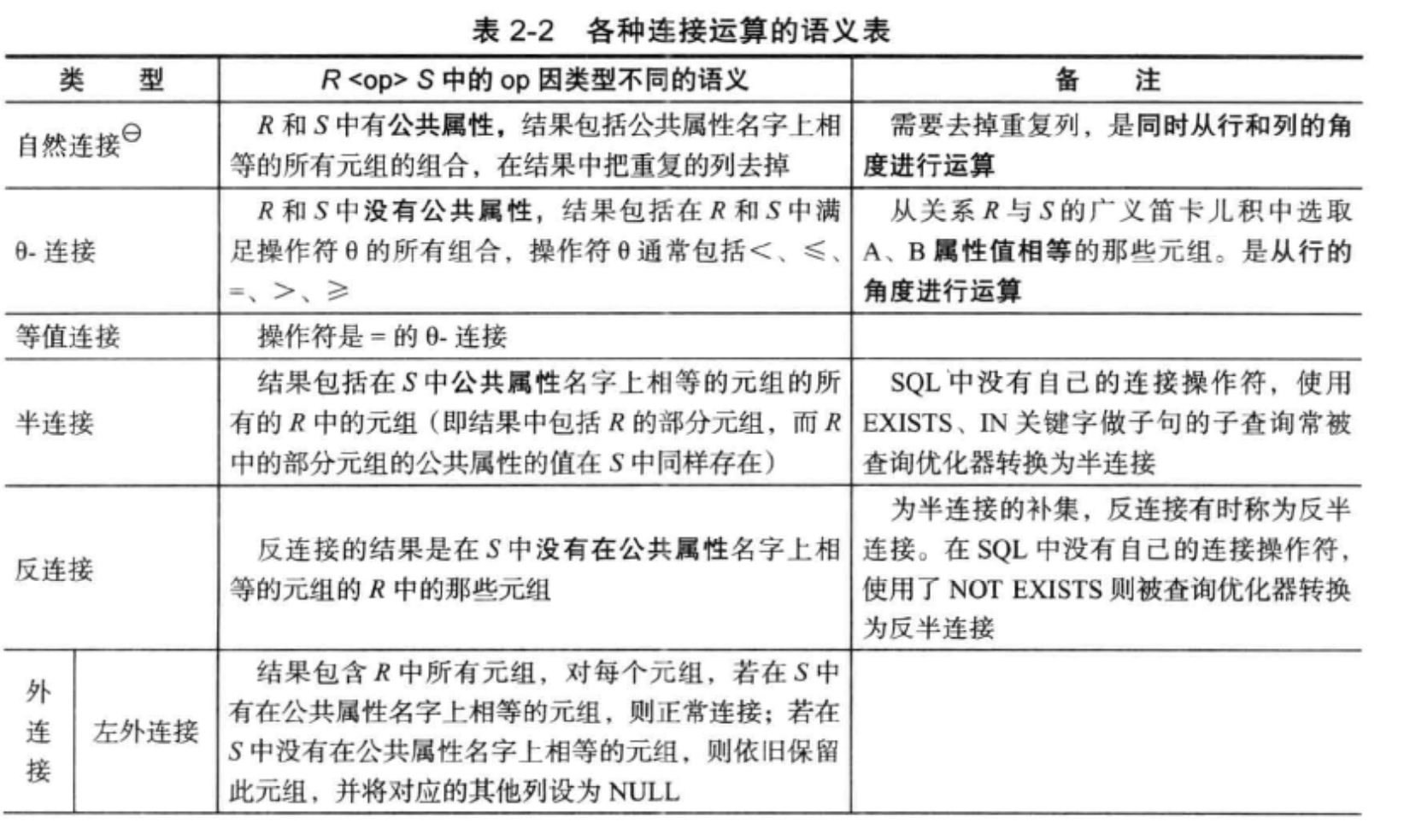
**专门的关系运算符**。选择（SELECT）、投影（PROJECT）、连接（JOIN）、除（DIVIDE）。

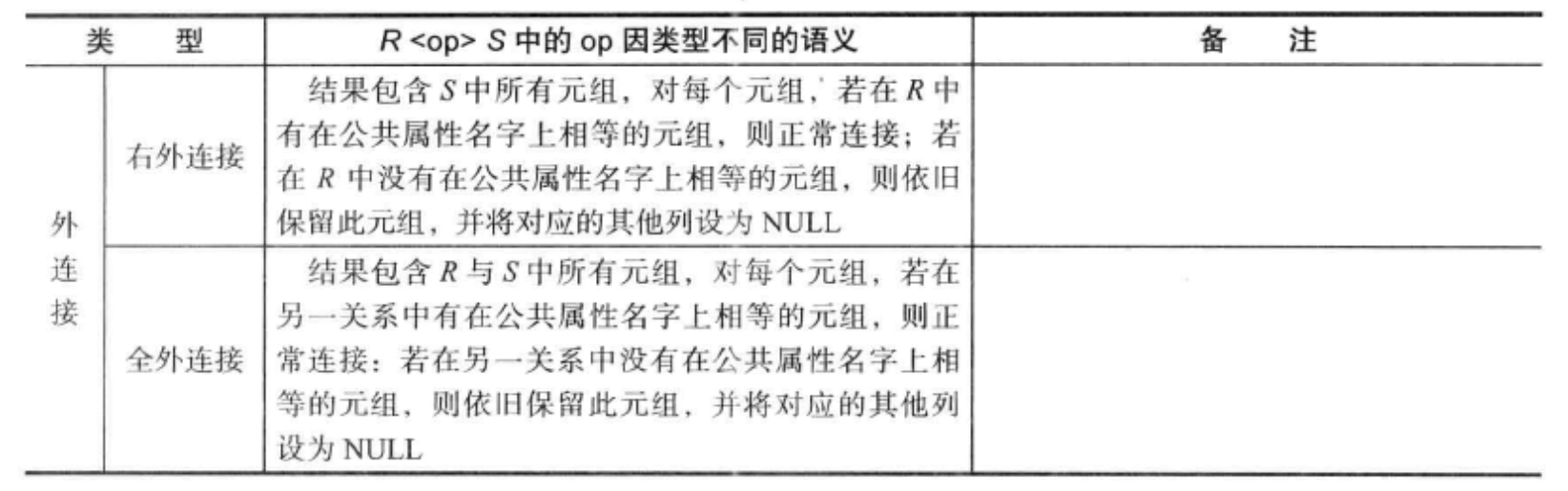
**辅助运算符**。用来辅助专门的关系运算符进行操作的，包括算术比较符合逻辑运算符。

Codd定义了8个基本关系运算后，许多人提出了新的代数操作符——**关系扩展运算符**。如半连接（SEMIJOIN）、半差（SEMIDIFFERENCE）、扩展（EXTEND）、合计（COMPOSITION）、传递闭包（TCLOSE）等。

### 运算语义







### 关系代数等价变换规则对优化的指导意义

## 查询重写规则

传统的联机事务处理（On-line Transaction Processing，OLTP）使用基于选择（SELECT）、投影（PROJECT）、连接（JOIN）3种基本操作相结合的查询，这种查询称为SPJ查询。

数据库在查询优化的过程中，会对这3种基本操作进行优化。优化的方式如下：

❏选择操作。对应的是限制条件（格式类似field＜op＞consant，field表示列对象，op是操作符，如=、＞等），优化方式是选择操作下推，目的是尽量减少连接操作前的元组数，使得中间临时关系尽量少（元组数少，连接得到的元组数就少），这样可减少IO和CPU的消耗，节约内存空间。

❏投影操作。对应的SELECT查询的目的列对象，优化方式是投影操作下推，目的是尽量减少连接操作前的列数，使得中间临时关系尽量小（特别注意差别：选择操作是使元组的个数“尽量少”，投影操作是使一条元组“尽量小”），这样虽然不能减少IO（多数数据库存储方式是行存储，元组是读取的最基本单位，所以要想操作列则必须读取一行数据），但可以各减少连接后的中间关系的元组大小，节约内存空间。

❏连接操作。对应的是连接条件（格式类似field\_1＜op＞field\_2，field\_1和field\_2表示不同表上的列对象，op是操作符，如=、＞等），表示两个表连接的条件。这里涉及以下两个子问题。

○多表连接中每个表被连接的顺序决定着效率。如果一个查询语句只有一个表，则这样的语句很简单；但如果有多个表，则会涉及表之间以什么样的顺序连接效率最高效（如A、B、C三表连接，如果ABC、ACB、BCA等连接后的结果集一样，则计算哪种连接次序的效率最高，是需要考虑的问题）。

○多表连接每个表被连接的顺序由用户语义决定。查询语句多表连接有着不同的语义（如是笛卡儿集、内连接，还是外连接中的左外连接等），这决定着表之间的前后连接次序是不能随意更换的，否则，结果集中数据是不同的。因此，表的前后连接次序是不能随意交换的。

另外，根据SQL语句的形式特点，还可以做如下区分：

❏针对SPJ的查询优化。基于选择、投影、连接3种基本操作相结合的查询。

❏针对非SPJ的查询优化。在SPJ的基础上存在GROUPBY操作的查询，这是一种较为复杂的查询。

所以，针对SPJ和非SPJ的查询优化，其实是对以上多种操作的优化。“选择”和“投影”操作，可以在关系代数规则的指导下进行优化。表连接，需要多表连接的相关算法完成优化。其他操作的优化多是基于索引和代价估算完成的。

### 子查询优化

从子查询出现在SQL语句的位置看，它可以出现在目标列（即select list）、where子句、join/on子句、group by子句、having子句、order by子句等位置。子查询出现在不同位置对优化的影响如下：

**目标列位置**。子查询如果位于目标列，则只能是标量子查询，否则数据库可能返回类似“错误：子查询必须只能返回一个字段”的提示。

**FROM子查询位置**。相关子查询出现在FROM子句汇总，数据库可能返回类似“在FROM子句中的子查询无法参与相同查询级别中的关系”的提示，所以相关子查询不能出现在FROM子句中；非相关子查询出现在FROM子句中，可上拉子查询到父层，在多表连接时统一考虑连接代价后择优。

**WHERE子句位置**。出现在WHERE子句中的子查询是一个条件表达式的一部分，而表达式可以分解为操作符和操作数；根据参与运算的数据类型的不同，操作符也不尽相同，如INT类型有>、<、=、<>等操作，这对子查询均有一定的要求（如INT类型的等值操作，要求子查询必须是标量子查询）。另外，子查询出现在WHERE子句中的格式也有用谓词指定的一些操作，如IN、BETWEEN、EXISTS等。

**JOIN/ON子句位置**。JOIN/ON子句可以拆分为两部分，一是JOIN块类似于FROM子句，二是ON子句类似于WHERE子句，这两部分都可以出现子查询。子查询的处理方式同FROM子句和WHERE子句。

**GROUP BY子句位置**。目标列必须和GROUP BY关联。可将子查询写在GROUP BY位置处，但子查询用在GROUP BY处没有实用意义。

**ORDER BY子句位置**。可将子查询写在ORDER BY位置处。但ORDER BY操作时作用在整条SQL语句上的，子查询用在ORDER BY处没有实用意义。

#### 分类

根据子查询中涉及的关系对象与外层关系对象间的关系，子查询可以分为以下两类：

**相关子查询**。子查询的执行依赖于外层父查询的一些属性值。子查询因依赖于父查询的参数，当父查询的参数改变时，子查询需要根据新参数值重新执行。

**非相关子查询**。子查询的执行不依赖于外层父查询的任何属性值，这样的子查询具有独立性，可独自求解，形成一个子查询计划先于外层的查询求解。

从特定谓词看，子查询可以分为以下三类：

[NOT]IN/ANY/ALL/SOME子查询。

[NOT]EXISTS子查询。

其他子查询。

从语句的构成复杂程度看，子查询可分为以下三类：

SPJ子查询。由选择、投影、连接操作组成的查询。

GROUP BY子查询。SPJ子查询加上分组、聚集操作组成的查询。

其他子查询。GROUP BY子查询中加上其他子句如Top-N、LIMIT/OFFSET、集合、排序等操作。后两种子查询有时候合成非SPJ子查询。

从结果集的角度看，子查询分为以下四类：

标量子查询。子查询返回的结果集类型时一个单一值（return a scalar，a single value）。

列子查询。子查询返回的结果集类型是一条单一数组（return a single row）。

行子查询。子查询返回的结果集类型时一个单一列（return a single column）。

表子查询。子查询返回的结果集类型是一个表（多行多列）（return a table，one or more row of one or more columns）。

#### 优化思路

子查询优化技术的思路如下：

**子查询合并**。在某些条件下（语义等价：两个查询块产生同样的结果集），多个子查询能够合并成一个子查询（合并后还是子查询，以后可以通过其他技术消除子查询）。这样可以把多次扫描、多次连接减少为单次表扫描和单次连接。

**子查询展开**。又称为查询反嵌套，又称为**子查询上拉**。把一些子查询置于外层的父查询中，作为连接条件与外层父查询并列，其实质是把某些子查询重写为等价的多表连接操作（展开后，子查询不存在了，外层查询变成了多表连接）。带来的好处是，有关的访问路径、连接方法和连接顺序可能被有效利用，使得查询语句的层次尽可能减少。常见的IN/ANY/SOME/ALL/EXISTS依据情况转换为半连接（SEMI JOIN）、普通类型的子查询消除等情况属于此类。

**聚集子查询消除**。聚集函数上推，将子查询转变为一个新的不包含聚集函数的子查询，并与父查询的部分或者全部表做左外连接。通常，一些系统支持的标量聚集子查询消除。

其他。利用窗口函数消除子查询技术、子查询推进等技术可用于子查询的优化。

子查询展开是一种最为常用的子查询优化技术，子查询展开有以下两种形式：

1. 如果子查询出现了聚集、GROUP BY、DISTINCT子句，则子查询只能单独求解，不可以上拉到上层；
2. 如果子查询只是一个简单格式（SPJ格式）的查询语句，则可以上拉到上层，这样往往能提高查询效率，子查询上来就是这种格式，这也是子查询展开技术处理的范围。

#### 常见子查询优化

##### IN类型

##### ANY/ALL/SOME类型

##### EXISTS类型

### 视图重写

视图是数据库中基于表的一种对象，视图重写就是将对视图的引用重写为对基本表的引用。**视图重写后的SQL多被作为子查询进行进一步优化**。所有的视图都可以被子查询替换，但不是所有的子查询都可以用视图替换。这是因为，子查询的结果作为一个结果集，如果是单行单列（标量），则可以出现在查询语句的目标列；如果是多行多列，可以出现在FROM、WHERE等子句中。但即使是标量视图（视图等同于表对象），也不可以作为目标列单独出现在查询语句中。

从视图的构成形式看，类似于查询的SPJ与非SPJ，视图可以分为简单视图和复杂视图。

用SPJ格式构造的视图，称为简单视图。

用非SPJ格式构造（带有GROUP BY等操作）的视图，称为复杂视图。

视图重写举例：

create table t1(a int key,b int);

create view v1 as select \* from t1;

基于视图的查询命令如下：

select a from v1 where b>100;

经过视图重写后可变换为如下形式：

select a from (select a,b from t1) where b>100;

简单视图能够被查询优化器较好地处理；但是复杂视图则不能被查询优化器很好地处理。一些商业数据库，如Oracle，提供了一些视图的优化技术，如“复杂视图合并”、“物化视图查询重写”等。但从整体上看，复杂视图优化技术还有待继续提高。

### 等价谓词重写

数据库执行引擎对一些谓词处理的效率要高于其他谓词，基于这点，把逻辑表达式重写成等价的且效率更高的形式，能有效提高查询执行效率。这就是等价谓词重写。

常见的等价谓词重写规则如下：

#### LIKE规则

LIKE谓词是SQL标准支持的一种模式匹配比较操作，LIKE规则是对LIKE谓词的等价重写，即改写LIKE谓词为其他等价的谓词，以更好地利用索引进行优化。

name LIKE ‘Abc%’

重写为：

name>=’Abc’ AND name<’Abd’

应用LIKE规则的好处是：**转换前针对LIKE谓词只能进行全表扫描，如果name列上存在索引，则转换后可以进行索引范围扫描**。

LIKE其他形式还可以转换：LIKE匹配的表达式中，若没有匹配符（%或\_），则与=等价。

例如：

name LIKE ‘Abc’

重写为：

name = ‘Abc’

#### BETWEEN-AND规则

BETWEEN-AND谓词是SQL标准支持的一种范围比较操作，BETWEEN-AND规则时指BETWEEN-AND谓词的等价重写，即改写BETWEEN-AND谓词为其他等价的谓词，以更好地利用索引进行优化。BETWEEN-AND谓词的等价重写类似于LIKE谓词的等价重写。

例如：

sno BETWEEN 10 AND 20

重写为：

sno>=10 AND sno<=20

应用BETWEEN-AND规则的好处是：如果sno上建立了索引，则可以用索引扫描代替原来BETWEEN-AND谓词限定的全表扫描，从而提高查询的效率。

#### IN转换OR规则

IN是指IN操作符操作，不是IN子查询。IN转换OR规则就是IN谓词的OR等价重写，即改写IN谓词为等价的OR谓词，以更好地利用索引进行优化。将IN谓词等价重写为若干个OR谓词，可能会提高执行效率。

例如：

age IN (1,2,3)

重写为：

age=1 OR age=2 OR age=3

应用IN转换OR规则后效率是否能够提高，需要看数据库对IN谓词是否只支持全表扫描。如果数据库对IN谓词只支持全表扫描且OR谓词中表的age列上存在索引，则转换后查询效率会提高。

#### IN转换ANY规则

IN转换ANY规则就是IN谓词的ANY等价重写，即改写IN谓词为等价的ANY谓词。因为IN可以转换为OR，OR可以转为ANY，所以可以直接把IN转换为ANY。将IN谓词等价重写为ANY谓词，可能会提高执行效率。

例如：

age IN (1,2,3)

重写为：

age ANY(1,2,3)

应用IN转换ANY规则后效率是否能够提高，依赖于数据库对于ANY操作的支持情况。如，PostgreSQL没有显式支持ANY操作，但是在内部实现时把IN操作转换为了ANY操作。

#### OR转换ANY规则

OR转换ANY规则就是OR谓词的ANY等价重写，即改写OR谓词为等价的ANY谓词，以更好地利用MIN/MAX操作进行优化。

例如：

sal>1000 OR dno=3 AND (sal>11—OR sal>base\_sal+100) OR sal>base\_sal+200 OR sal>base\_sal\*2

重写为：

dno=3 AND (sal>1100 OR sal>base\_sal+100) OR sal>ANY(1000,base\_sal+200,base\_sal\*2)

OR转换ANY规则依赖于数据库对于ANY操作的支持情况。

#### ALL/ANY转换集函数规则

ALL/Any转换集函数规则就是将ALL/ANY谓词改写为等价的聚集函数MIN/MAX谓词操作，以更好地利用MIN/MAX操作进行优化。

例如：

sno>ANY(10,2\*5+3,sqrt(9))

重写为：

sno>sqrt(9)

注意：

1. 本示例中存在>和ANY，其意是在找出(10,2\*5+3,sqrt(9))中的最小值，所以可以重写为sno>sqrt(9)
2. 通常，聚集函数MAX()、MIN()等的执行效率比ANY、ALL谓词的执行效率高，因此在这种情况下对其进行重写可以起到比较好的效果。如果有索引存在，求解MAX/MIN的效率更高。

#### NOT规则

NOT谓词的等价重写。如下：

NOT (col1!=2)重写为col1=2

NOT (col1!=col2)重写为col1=col2

NOT (col1=2)重写为col1!=col2

NOT (col1<col2)重写为col1>=col2

NOT (col1>col2)重写为col1<=col2

NOT规则重写的好处是：如果在col1上建立索引，则可以用索引扫描代替原来的全表扫描，从而提高查询的效率。

#### OR重写并集规则

OR条件重写为并集操作。

例如：

SELECT \* FROM student WHERE (sex=’f’ AND sno>15) OR age>18;

假设所有条件表达式的列上都有索引（即sex列和age列上都存在索引），数据库可能示例中的WHERE语句强迫查询优化器使用顺序存取，因为这个语句要检索的是OR操作的集合。为了能利用索引处理上面的查询，可以将语句改成如下形式：

SELECT \* FROM student WHERE sex=’f’ AND sno>15 UNION

SELECT \* FROM student WHERE age>18

改写后的形式，可以分别利用列sex和age上的索引，进行索引扫描，然后再提供执行UNION操作获得最终结果。

### 条件化简

WHERE、HAVING和ON条件由许多表达式组成，而这些表达式在某些时候彼此之间存在一定的联系。利用等式和不等式的性质，可以将WHERE、HAVING和ON条件化简，不同数据库的实现可能不完全相同。

将WHERE、HAVING和ON条件化简的方式通常包括如下几个：

**将HAVING条件并入WHERE条件。**便于统一、集中化条件子句，节约多次化解时间。但并不是任何情况下HAVING条件都可以并入WHERE条件，只有在SQL语句中不存在GROUP BY条件或聚集函数的情况下，才能将HAVING条件与WHERE条件的进行合并。

**去除表达式中冗余的符号**。这样可以减少语法分析时产生的AND和OR树的层次。如((a AND b) AND (c AND d)就可以化简为a AND b AND c AND d。

**常量传递**。对不同关系可以使得条件分离后有效实施“选择下推”，从而可以极大地减小中间关系的规模。如col1=col2 AND col2=3就可以化简为col1=3 AND col2=3。操作符=、<、>、<=、>=、<>、LIKE中的任何一个，在col1<操作符>col2条件中都会发生常量传递。

**消除死码**。化简条件，将不必要的条件去除。如WHERE (0>1 AND s1=5)，0>1使得AND恒为假，则WHERE条件恒为假。此时就不必再对该SQL语句进行优化和执行了，加快了查询执行的效率。

**表达式计算**。对可以求解的表达式进行计算，得出结果。如WHERE col1=1+2变换为WHERE col=3。

**等式变换**。化简条件（如反转关系操作符的操作数的顺序），从而改变某些表的访问路径。如-a=3可化简为a=-3.这样的好处是如果a上有索引，则可以利用索引扫描来加快访问。

**不等式变换**。化简条件，将不必要的重复条件去除。如a>10 AND b=6 AND a>2可化简为b=6 AND a>10。

**布尔表达式变换**。布尔表达式还有如下规则指导化简。

1. 谓词传递闭包。一些比较操作符，如<、>等，具有传递性，可以起到化简表达式的作用。如由a>b AND b>2可以推导出a>b AND b>2 AND a>2，a>2是一个隐含条件，这样把a>2和b>2分别下推到对应的关系上，就可以减少参与比较操作a>b的元组了。
2. 任何一个布尔表达式都能被转换为一个等价的合取范式（CNF）。因为合取项只要有一个为假，整个表达式就为假，故代码中可以在发现一个合取项为假时，即停止其他合取项的判断，以加快判断速度。另外，因为AND操作符时可交换的，所以优化器可以按照先易后难的顺序计算表达式，一旦发现一个合取项为假时，即停止其他合取项的判断，以加快判断速度。
3. 索引的利用。如果一个合取项上存在索引，则先判断索引是否可用，如能利用索引快速得出合取项的值，则能加快判断速度。同理，OR表达式中的子项也可以利用索引。

### 外连接消除

外连接消除的意义

外连接操作可分为左外连接、右外连接和全外连接。连接过程中，外连接的左右子树不能互换，并且外连接与其他连接交换连接顺序时，必须满足严格的条件以进行等价变换。这种性质限制了优化器在选择连接顺序时能够考虑的表与表交换连接位置的优化方式。

查询重写的一项技术就是把外连接转为内连接，转换的意义（对优化的意义）如下：

查询优化器在处理外连接操作时所需要执行的操作和时间多于内连接。

优化器在选择表连接顺序时，可以有更多更灵活的选择，从而可以选择更好的表连接顺序，加快查询执行的速度。

表的一些连接算法（如块嵌套连接和索引循环连接等）将规模小的或者筛选条件最严格的表作为“外表”（放在连接顺序的最前面，是多层循环体的外循环层），可以减少不必要的IO开销，极大地加快算法执行的速度。

**外连接消除的条件**

外连接可转换为内连接的条件：WHERE子句中与内表相关的条件满足：空值拒绝（reject-NULL条件）。一般认为满足下面任意一种情况时，即满足空值拒绝。

1. 条件可以保证从结果中排除外连接右侧（右表）生成的值为NULL的行（即条件确保应用在右表带有空值的列对象上时，条件不满足，条件的结果值为FALSE或UNKNOWEN，这样右表就不会有值为NULL的行生成），所以能使该查询在语义上等效于内连接。
2. 外连接的提供空值的一侧（可能是左侧的外表也可能是右侧的内表）为另一侧的每行只返回一行。如果该条件为真，则不存在提供空值的行，并且外连接等价于内连接。

### 嵌套连接消除

多表连接有时会存在嵌套的情况。对于一个无嵌套的多表连接，表之间的连接次序是可以交换的，这样能灵活求解不同连接方式的花费，进而得到最小花费的连接方式。而嵌套连接则不能够利用交换表的位置而获得优化。

当执行连接操作的次序不是从左到右逐个进行时，就说明这样的连接表达式存在嵌套。

例如：

SELECT \* FROM t1 LEFT JOIN (t2 LEFT JOIN t3 ON t2.b=t3.b) ON t1.a=t2.a WHERE t1.a>1

先t2与t3连接，得到中间结果{t2 t3}后再与t1连接，这种方式就是嵌套连接，括号不可以去掉，没有去掉括号的等价形式。如果连接顺序时t1、t2、t3，则不存在嵌套。

另外，如下格式也是嵌套，这种格式用括号把连接次序做了区分。

SELECT \* FROM A JOIN (B JOIN C ON B.b1=C.c1) ON A.a1=B.b1 WHERE A.a1>1;

上面的格式可以等价转换为（圆括号去掉，不影响原来语义）下面的格式：

SELECT \* FROM A JOIN B JOIN C ON B.b1=C.c1 ON A.a1=B.b1 WHERE A.a1>1;

综上所述，得到结论：

1. 如果连接表达式只包括内连接，括号可以去掉，这意味着表之间的次序可以交换，这是关系代数中连接的交换律的应用。
2. 如果连接表达式包括外连接，括号不可以去掉，意味着表之间的次序只能按照原语义进行，至多能执行的就是外连接向内连接转换的优化。

### 连接消除

### 语义优化

### 针对非SPJ优化

#### GROUP BY优化

对于GROUP BY的优化，可以考虑分组转换技术，即对分组操作、聚集操作与连接操作的位置进行变换。常见的方式如下：

**分组操作下移**。GROUP BY操作可能较大幅度地减少关系元组的个数，如果能够对某个关系先进行分组操作，然后再进行表之间的连接，很可能提高连接效率。这种优化方式是把分组操作提前执行。**下移的含义，是在查询树上让分组操作尽量靠近叶子节点，使得分组操作的节点低于一些选择操作**。

**分组操作上移**。如果了解操作能够过滤掉大部分元组，则先进行连接后进行GROUP BY操作，可能提高分组操作的效率。这种优化方式是把分组操作置后执行。

GROUP BY操作下移或者上移均不能保证重写后的查询效率一定更好，所以，要在查询优化器中采用基于代价的方式来估算几种路径的优劣。

另外，GROUP BY、ORDER BY优化的另外一个思路是尽量利用索引。

#### ORDER BY优化

对于ORDER BY的优化，可有如下方面的考虑：

**排序消除（Order By Elimination，OBYE）**。优化器在生成执行计划前，将语句中没有必要的排序操作消除（如利用索引），避免在执行计划中出现排序操作或由排序导致的操作（如在索引列上排序，可以利用索引消除排序操作）。

**排序下推（Sort push down）**。把排序操作尽量下推到基表中，有序的基表进行连接后的结果符合排序的语义，这样能避免在最终的大的连接结果集上执行排序操作。

#### DISTINCT优化

对于DISTINCT的优化，可有如下方面的考虑：

**DISTINCT消除（Distinct Elimination）**。如果表中存在主键、唯一约束、索引等，则可以消除查询语句中的DISTINCT（这种优化方式，在语义优化中也涉及，本质上是语义优化研究的范畴）。

**DISTINCT推入（Distinct Push Down）**。生成含有DISTINCT的反半连接查询执行计划时，先进行反半连接再进行DISTINCT操作，也许先执行DISTINCT操作再执行反半连接更优，这是利用连接语义上确保唯一功能特性进行DISTINCT的优化。

**DISTINCT迁移（Distinct Placement）**。对连接操作的结果执行DISTINCT，可能把DISTINCT移到一个子查询中优先进行。

## 启发式规则在逻辑优化阶段的应用

逻辑优化阶段使用的启发式规则通常包括如下两类。

1. 一定能带来优化效果的，主要包括：

优先做选择和投影（连接条件在查询树上下推）。

子查询的消除。

嵌套连接的清除。

外连接的清除。

连接的清除。

使用等价谓词重写对条件化简。

语义优化。

裁剪冗余操作（一些剪枝优化技术）、最小化查询块。

1. 变换未必会带来性能的提高，续根据代价选择，主要包括：

分组的合并。

借用索引优化分组、排序、DISTINCT等操作。

对视图的查询变为基于表的查询。

连接条件的下推。

分组的下推。

连接提取公共表达式。

谓词的上拉。

用连接取代集合操作。

用UNION ALL取代OR操作。

# 物理查询优化

## 查询代价估算

## 单表扫描算法

## 索引

## 两表连接算法

## 多表连接算法

多表连接算法实现的是在查询路径生成的过程中，根据代价估算，从各种可能的候选路径中找出最优的路径（最优路径是代价最小的路径）。

多表连接算法需要解决两个问题：

**多表连接的顺序**：表的不同的连接顺序，会产生许多不同的连接路径；不同的连接路径有不同的效率。

多表连接的搜索空间：因为多表连接的顺序不同，产生的连接组合会有多种，如果这个组合的数目巨大，连接次数会达到一个很高的数量级，最大可能的连接次数是N!。所有的连接可能构成一个巨大的“搜索空间”。如何将搜索空间限制在一个可接受的时间范围内，并高效地生成查询执行计划将成为一个难点。

表与多进行连接，对多表连接进行搜索查找最优查询树，通常有多种算法，比如启发式、分枝界定计划枚举、爬山法、动态规划、System R优化方法等。

# 查询优化器原理解析

## 概述

MySQL查询优化器主要的功能是完成SELECT语句的执行，在保证SELECT语句正确执行之外，还有一个重要的功能，就是使用关系代数、启发式规则、代价估算模型等不同种类的技术，提高SELECT语句执行的效率。

### 查询执行过程

MySQL查询执行过程分为4个阶段，如下所示。

❏语法分析阶段：将SQL查询语句经词法和语法分析后变换为一棵查询树st\_select\_lex传给优化器，并对SQL表达的语义进行检查。

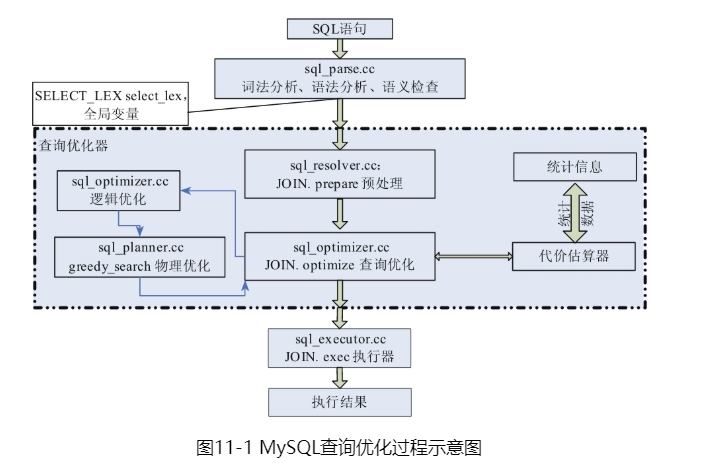
❏生成逻辑查询执行计划阶段：优化器在查询树中遍历每个关系，确定关系是否是常量表、为每个关系查找可用的索引、运用关系代数原理和启发式规则进行逻辑上的查询优化（如消除子查询、消除外连接等）。

❏生成物理查询执行计划阶段：优化器对各个连接的表进行排序，然后再求解多表连接最优路径，对于每个关系尽量利用索引计算其代价，找出代价最小的路径后保存到JOIN类的best\_positions。

❏执行查询执行计划阶段：把查询执行计划传到执行器进行执行。4个阶段如图11-1所示。

MySQL查询优化器在逻辑查询执行计划阶段，基于关系代数规则和启发式规则，把用户指定的SQL经过“等价”的代数转换，变为一种更节省IO的执行序列，执行起来更为高效。

MySQL查询优化器在物理查询执行计划阶段，在解决多表连接的问题时，有两套算法：一是用户指定表连接次序的算法；二是混杂了贪婪和穷举思想的算法，解决的是较多表的连接或非用户指定连接次序的多表连接，但不能保证得到最优的查询执行计划。



### 设计思想

MySQL查询优化的过程中，查询优化器通过JOIN对象的方法，如JOIN::prepare()、JOIN::optimize()，完成优化工作。JOIN::prepare()完成的查询优化主要包括：子查询的冗余子句消除、IN类型子查询优化、将ALL/Any等类型的子查询转换为MIN/MAX等操作，这是对简单子查询进行的优化；JOIN::optimize()函数完成的查询优化主要包括：子查询上拉，把外连接优化为内连接，把嵌套连接消除，WHERE子句、JOIN/ON子句、HAVING子句条件表达式的化简（尤其是对含有常量的表达式的化简、等式合并），优化没有GROUP BY子句情况下COUNT(\*)、MIN()、MAX()、裁剪分区partition（如果查询的表是分区表），确定多表的连接路径（单表是多表的特例，统计join的代价，两种多表连接算法选其一搜索最有的join顺序、生成执行计划）、优化等式谓词、优化DISTINCT、创建临时表存储临时结果优化分组排序等操作。在这样的过程中，MySQL没有把优化过程明显地分为逻辑查询优化阶段和物理查询优化阶段，而是互相混杂。

### 主要概念

#### 常量表

MySQL中常量表的“常量”不等同编程语言中指定的常量。在MySQL中，常量表是指读取表上的数据行数是确定的零行或一行，分为两种类型：

System表：表中只有一行数据。

Const表：经where语句中的限制条件（“column=常数”格式的表达式）筛选后只有一行数据；这表明，在column列上存在索引，索引列可能是一个主键列，也可能是唯一键的列。

#### 表数据访问方式

### 代码层次



## MySQL查询优化器相关数据结构

### 查询树

语法分析的结果是查询树，使用st\_select\_lex/SELECT\_LEX类表示，类的实例保存了一条SQL语句被分解后的各个子句的内容。st\_select\_lex类继承自st\_select\_lex\_node类，获得了父类的一些基本信息，如e\_sql\_cache\_sql\_cacheSQL缓存信息。

注：在MySQL5.5中查询树的类是st\_select\_lex，MySQL5.6中查询树的类是st\_select\_lex（继承自Sql\_alloc）MySQL8.0中改为SELECT\_LEX。

### 基本对象

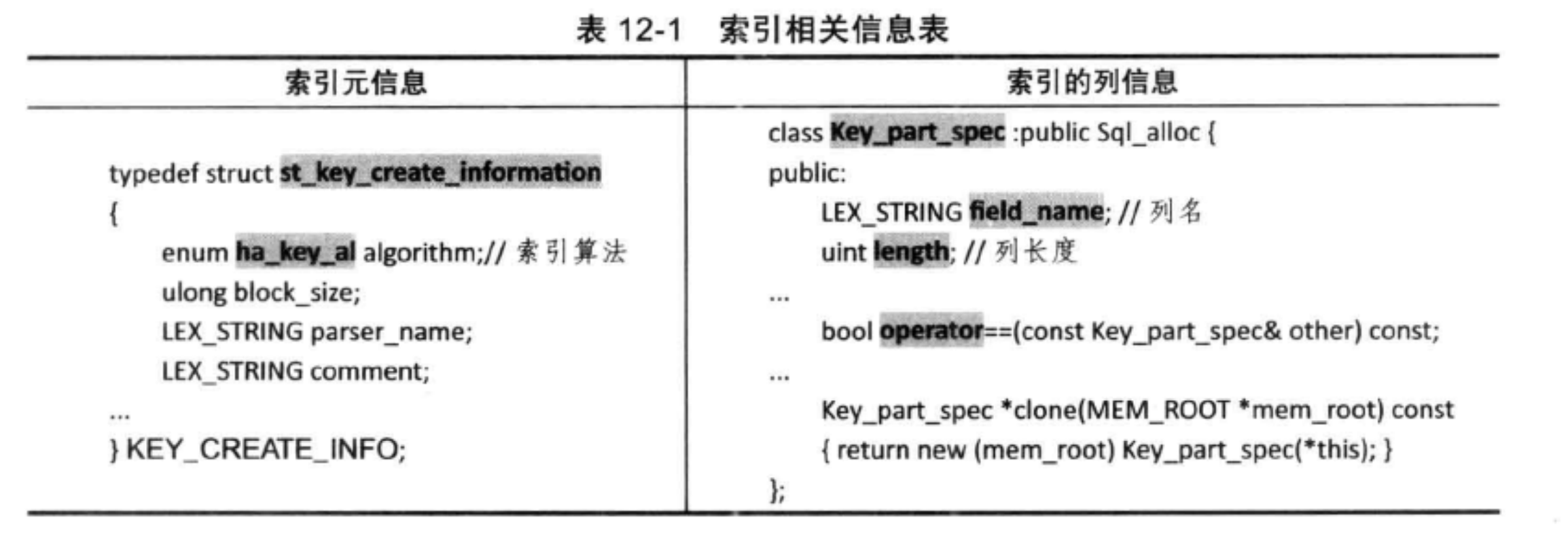
MySQL的基本对象有很多，和查询优化相关的主要由关系（TABLE\_LIST）、索引（Key）、连接表（JOIN\_TAB）等。

#### 关系

MySQL在一个关系（表）上存放了很多信息，有查询优化阶段需要用到的信息，也有其他阶段用到的信息。**基本上，只要和表相关的信息都存放在TABLE\_LIST这个结构体中，这是逻辑上的表的结构，在存储层，由TABLE结构体表示表对象**。

#### 索引

MYSQL通过Key类，定义了索引的类型和索引的关键信息。其中的KEY\_CREATE\_INFO包括了索引列、索引算法等。



注：MySQL5.5/5.6索引类为class Key，MySQL8.0为class Key\_spec。

#### 连接表

连接表是介于关系（TABLE\_LIST类）与连接类（JOIN类）之间的一个过渡对象，存放了关系的一些相关信息，也存放了连接操作需要的一些信息，所以称为连接表。

对应的类：class JOIN\_TAB。

### 连接对象与执行计划

MySQL查询优化相关的连接对象主要有连接类（JOIN）、约束条件（Item）、位置（POSITION）等。

#### 连接类（JOIN）

JOIN类，连接操作对应的查询语句的连接关系内容，是优化和执行的基本单元，也是优化结果（查询执行计划）的存储对象。

#### 约束条件

约束条件是指WHERE或JOIN/ON或HAVING子句中的谓词表达式，其分为两种：一种是限制条件，用来过滤单表的元组；另一种是连接条件，满足连接条件的元组才会连接，连接条件表达式一般包括两个或两个以上关系的变量。

MySQL用Item类表示约束条件表达式，这是一个父类，由一系列的子类继承这个父类。

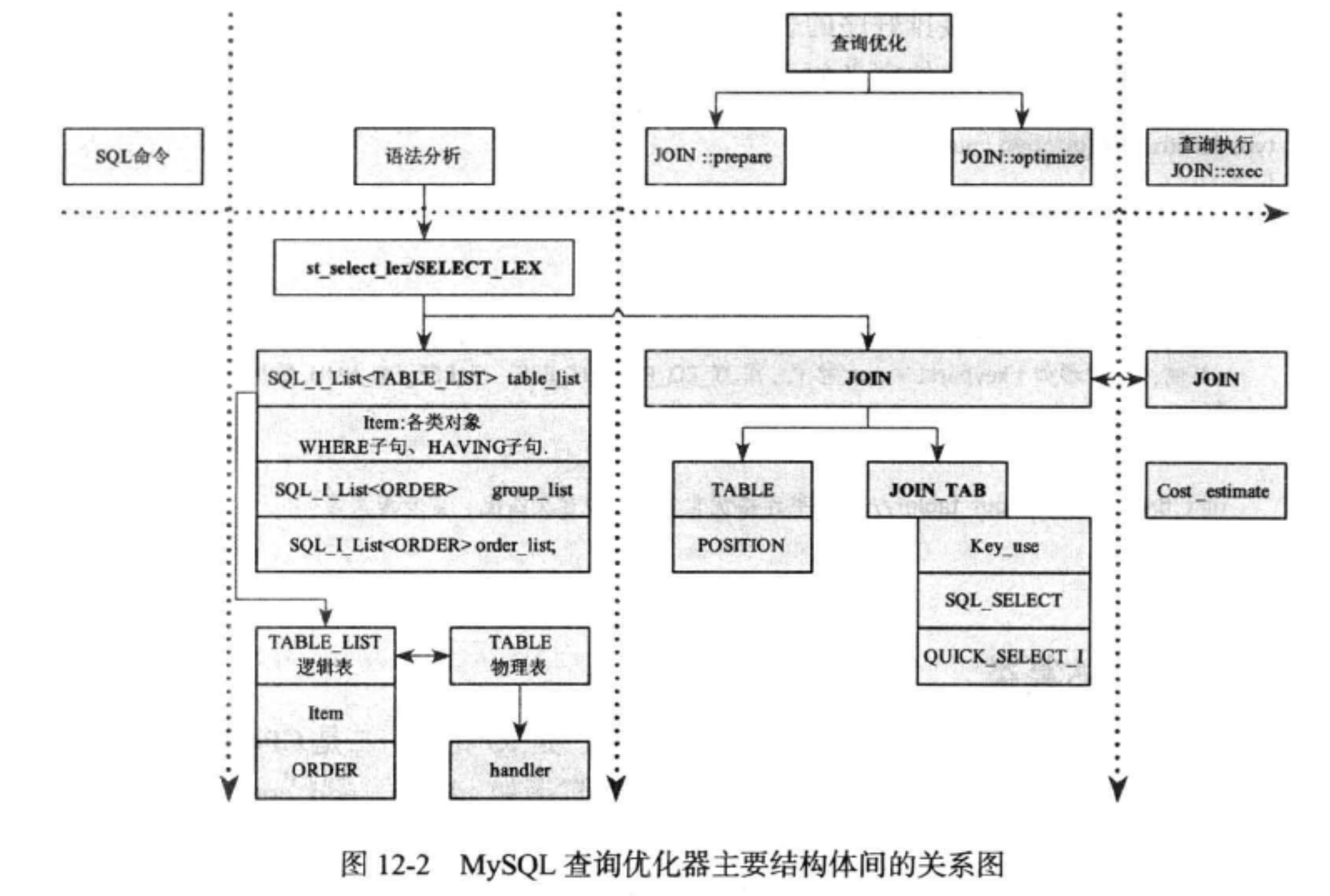
#### 位置

在连接顺序中，位置是指被连接的表的位置（一个表在多表连接构成的连接顺序中处于的位置）。这个位置上存了被访问的表、被使用的访问方法、半连接策略的选择、半连接优化状态等。MySQL的多表连接算法支持的是从左到右的类似左深树的连接方式，但左深树是一个树型结构，而MySQL的多表连接算法连接表的过程是一个平面结构，先对表进行排序，然后从左到右地连接排序好的表，然后确定当前连接方式的花费。

注：MySQL5.5/5.6使用的是结构体st\_position，MySQL8.0是类class POSITION。

### 代价估算类

### 各个阶段主要结构体间关系



## 原理

## 查询优化

## 关键算法