级联火山口：数据库查询优化器初探

Accela Zhao (20210409)

目录

复杂度的来源 2
基本名词 3
基于代价的优化器（Cost-based Optimizer）、基于规则的优化器（Rule-based Optimizer）、Heuristic-based optimizer 3
Selectivity、Cardinality 4
逻辑计划（Logical Plan）、物理计划（Physical Plan） 4
Operator、Logical Operator、Physical Operator 4
Volcano 和 Cascades 5
基本概念 – 三大组件 5
基本概念 – Operator 6
基本概念 – Pattern/Rule 6
基本概念 – Memo 7
基本概念 – 任务调度 8
Volcano 与 Cascades 的区别 9
搜索计划空间（Plan Enumeration） 9
局部最优问题 9
关于 Property 和 Enforcer 10
搜索优先级 11
避免重复搜索 11
剪枝（Pruning） 12
搜索退出条件 12
Join Order Enumeration 12
搜索算法的详细例子 12
Columbia 的详细例子 12
Memo 的详细例子 18
代价模型（Cost Model） 18
基础 Cost Variable 19
更复杂的代价模型 20
代价模型的分析和验证 22
统计信息（Statistics） 25
柱状图（Histogram） 25
Statistics Derivation 25
查询执行（Query Execution） 27

切分和平衡......27
NUMA 架构......28
分布式 Operator......28
总结......28
相关资料......28

查询优化器（Query Optimizer）是数据库的**标志技术**，也是其中最难懂、最少懂的部分（以及事务处理[54]）。它能够影响 SQL 执行速度达 2x~100x[14]（例如扫描 vs 索引、Join 实现的选择）。此外，

- 现代数据库**语言支持**往往远超 SQL[52]，数据种类多样，查询优化器有更大发挥空间；例如嵌入 Python/Java 语言，直接调用 Spark、Hadoop，组建数据处理管线（Pipeline/Dataflow）。
- 数据库对大数据、OLAP 、异构**集成**的支持，对(Continuous) Stream、Graph、机器学习的**集成**，也为查询优化器带来新的挑战。

数据库之外，查询优化器的设计可被学习至更多领域，例如

- 如何将 CSV 文件[35]当作数据表用 SQL 查询，如何用类 SQL 语言进行日志搜索。更丰富的工具引入更丰富的接口，直至领域**语言**（Domain Specific Language、DSL），产生语法解析和优化的需求。
- 代价模型（Cost Model）的设计[6]，如何计量建模 CPU、IO、网络开销，可应用于更多系统的**资源调度**，如分布式存储。
- 搜索巨大复杂**空间**的方法，例如 SQL 执行计划。更重要的是如何将其抽象成整洁易扩展的设计。

复杂度的来源

组合[33]是宇宙神秘之一，简单规则[51]即可突破宇宙原子数，空间结构[53]可研究但仍不足。

SQL 语句被翻译为关系代数（Relation Algebra）表达式后，查询优化器需在大量等价表达中寻找最优。逻辑（Logical）优化的典型例子是交换 Join 的顺序，见下表[10P27]，**组合增长迅速**。

Number of Relations	Number of Groups	Number of Logical Expressions	Number of Query Trees
N	$2^N - 1$	$3^N - (2^{N+1} - 1) + N$	$(2N - 2)! / (N - 1)!$
2	3	4	2
3	7	15	12
4	15	54	120
5	31	185	1680
6	63	608	30240
7	127	1939	665280
8	255	6058	17297280
9	511	18669	518918400
10	1023	57012	17643225600
11	2047	173063	670442572800
12	4095	523262	28158588057600
13	8191	1577953	1295295050649600
14	16383	4750216	64764752532480000
15	32767	14283387	3497296636753920000
16	65535	43915666	202843204931727360000
17	131071	128878037	12576278705767096254464
18	262143	386896220	830034394580628315045888

Table 1. Complexity of Join of n Relations [Bil97]

进一步，物理优化需选择 Join 实现算法，如 Nested-loop Join、Hash Join、Index Join、Sort-merge Join；更多组合。分布式数据库还需选择 Join 数据移动策略，如 Broadcast、Shuffle，甚至更细致的 Track Join[19]。（Join 组合如此之多，大部分优化器甚至只搜索 Left-deep Tree，而忽略 Bushy Tree）

除 Join 外，诸多其它 Operator 同样需要优化，如谓词下推（Predict Pushdown）。此外，优化搜索需要在有限时间完成，否则增加 SQL 整体执行时间；需要准确计算每种执行计划的代价；还需要维护代价相关的统计信息，尤其是不同模式的数据快速输入时。

基本名词

基于代价的优化器（Cost-based Optimizer）、基于规则的优化器（Rule-based Optimizer ）、Heuristic-based optimizer

它们的差异在于判断执行计划好坏的方法。

基于规则的优化器根据固定的、通常是手写的规则，决定哪个执行计划更好。例如当 Selectivity 小于 0.01 时使用索引，否则做全表扫描。Heuristic-based optimizer 一般与基于规则的优化器同义，Heuristic 指人工想出规则（启发式）。

基于代价的优化器实地计算各个执行计划的 CPU、IO 等代价（代价模型），然后比较好坏。显而易见，基于代价的优化器更加准确[6]，也更加复杂。现代优化器（通常以 Volcano/Cascades 为原型）和企业级数据库（Oracle、SQL Server 等），主流都是基于代价的优化器。

（此外，Rule-based、Cost-based 有时也用来区分查询计划遍历方法[14]，例如 Rule-based 用固定规则枚举 Join 顺序，而 Cost-based 通常配套动态规划搜索。）

实际上，基于代价的优化器常常结合基于规则的优化。首先做基于规则的优化，例如那些总是更优的转换（Transform）；然后对不确定的部分基于代价搜索。例如 CockroachDB 使用基于代价的优化器，但用声明式的 Optgen 领域语言[37]定义了许多规则转换。

Selectivity、Cardinality
Selectivity 指查询会从表中返回多大比例的数据。例如表有 1000 行（Tuple），Select where column < 10 返回 10 行，那么 Selectivity 是 1%。当 Selectivity 极高时，全表扫面通常比索引查询更高效。

Cardinality 的基本含义是表的 unique 行数[34]，查询计划中常指 Operator 需要处理的行数[36]。初始表的行数是叶节点 Operator 需处理的 Cardinality。Operator 的输出（例如 Join）可看作广义的表，它又是上游 Operator（下一个待处理的 Operator）的 Cardinality。Operator 输出输入的比例是 Selectivity。

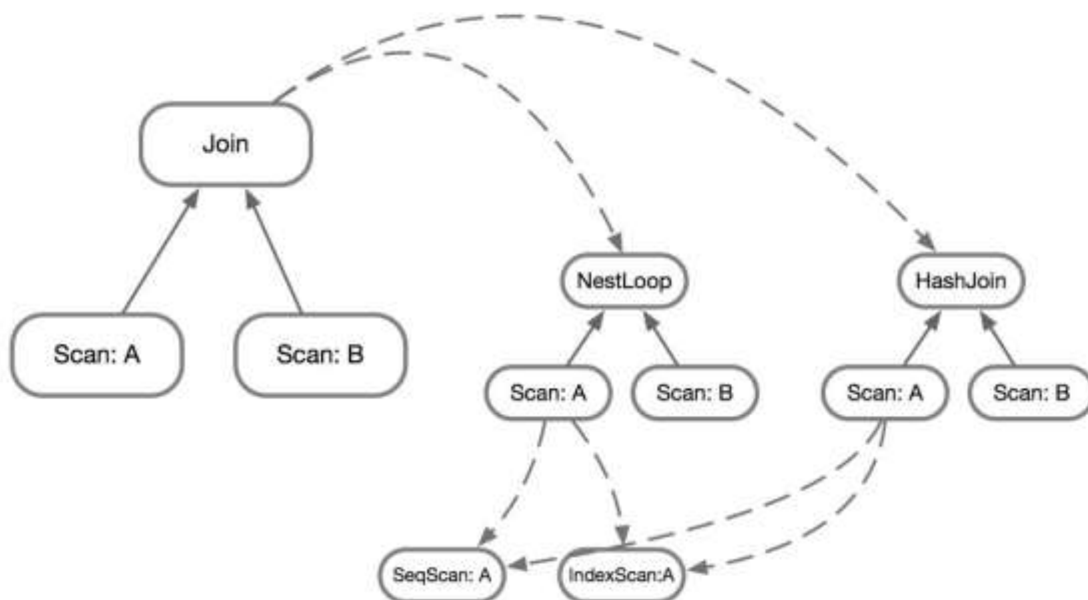
显而易见，Cardinality 估算（Cardinality Estimation）是为执行计划计算代价的核心[14]。

逻辑计划（Logical Plan）、物理计划（Physical Plan）
以"复杂度的来源"中的 Join 为例，交换 Join 的顺序属于逻辑优化，对应逻辑计划。将 Join 展开为 Nested-loop Join、Hash Join 等，属于物理优化，对应物理计划。

SQL 语句被翻译为关系代数表达式树后，可以看出 Operator 需执行的先后顺序；这就是逻辑计划。树中 Operator 是关系代数运算符，称作 Logical Operator。优化器需探索大量等价的表达式树，搜索逻辑计划空间，称作逻辑优化。逻辑优化通常只需基于规则的转换（Transform），不会将 Logical Operator 转换为 Physical Operator，不需计算代价。

一个 Logical Operator 通常可选择多种物理实现算法，即 Physical Operator。优化器会逐步将 Logical Operator 展开为 Physical Operator，根据代价模型选择优者。此时称作物理优化。最终表达式树完全由 Physical Operator 构成，组成一颗树（表达式树），可由执行引擎直接执行；称作物理计划，也作执行计划。

逻辑计划和物理计划例下图[1]。



（基于代价的优化器鼻祖 Volcano 论文中，逻辑优化和物理优化分为两个阶段。但后来 Cascades 论文中，逻辑优化和物理优化合二为一，以此避免探索无效的逻辑计划空间。因而今日"逻辑计划"与"物理计划"术语也有些混淆。）

Operator、Logical Operator、Physical Operator
逻辑、物理执行计划通常表达为树的形式，每个节点为 Operator。Operator 意指一个 SQL 操作，例如 Join、Filter（Select Predict）；逻辑上对应 Logical Operator，物理实现算法对应 Physical Operator；逻辑计划对应 Logical Operator，物理计划对应 Physical Operator。

Operator 执行时，以 Filter 为例，一面输入扫描表中行，一面输出满足 Filter 谓词的行。更多地，

- 可用迭代器 **Poll** 方式实现，父 Operator 调用 next 以使子 Operator 返回下一结果。
- 可用 **Push** 方式实现，子 Operator 凑齐结果后上推给父 Operator。
- 可用**向量化**[17]（Vectorized）方式实现，Operator 一次处理多行（Vector）而不是单行。
- 可用 **Compiled Query**[20]，将多 Operator 合并紧凑编译（Loop Fusion、Loop Pipelining）以提速。

Volcano 和 Cascades

Volcano[7]和 Cascades[8]源自如下两篇论文，它们是现代基于代价的优化器的鼻祖。如今各大企业级数据库和开源数据库的查询优化器仍采用其架构（如 TiDB[2]、CockroachDB[5]、Greenplum（Orca[12]）、Apache Calcite[11]）。

- The **Volcano** Optimizer Generator: Extensibility and Efficient Search
- The **Cascades** Framework for Query Optimization

Volcano 继承自更古老的 System-R、Exodus、Starburst。Cascades 升级 Volcano 做了诸多改进，二者出自同一作者（Goetz Graefe 等），很多文献把 Volcano/Cascades 名词混用。

Volcano/Cascades 的卓越之处在于为混沌复杂的查询优化器世界划定了优雅的**抽象框架**，后来者只需继承抽象类[2]就能构造新的查询优化器。

与其直接罗列 Volcano/Cascades 的概念，我们从解决设计问题的角度讲述。

基本概念 – 三大组件

在设计的开始，首先需要确定优化器的**输入、输出**，如下图[10P15]

- **输入**：SQL 文本首先被语法解析（Parse）为 AST（Abstract Syntax Tree），然后翻译成 Logical Operator 树（关系代数表达式，初始的逻辑计划），作为优化器的输入。因 SQL 语法成熟固定，这步并无太多讨论，可采用 BNF 范式状态机等。
- **输出**：代价最优的 Physical Operator 树，即物理计划；交由执行引擎执行，访问存储、汇聚结果。执行引擎如何切分并行化 Operator、如何分布式调度、如何向量化，大有文章，但不在查询优化器范围内。
- **元数据**：通常有外部服务提供数据库表目录（Catalog），提供有关代价的统计信息。元数据 Provider 甚至可以插件[11]提供，以兼容异构平台。

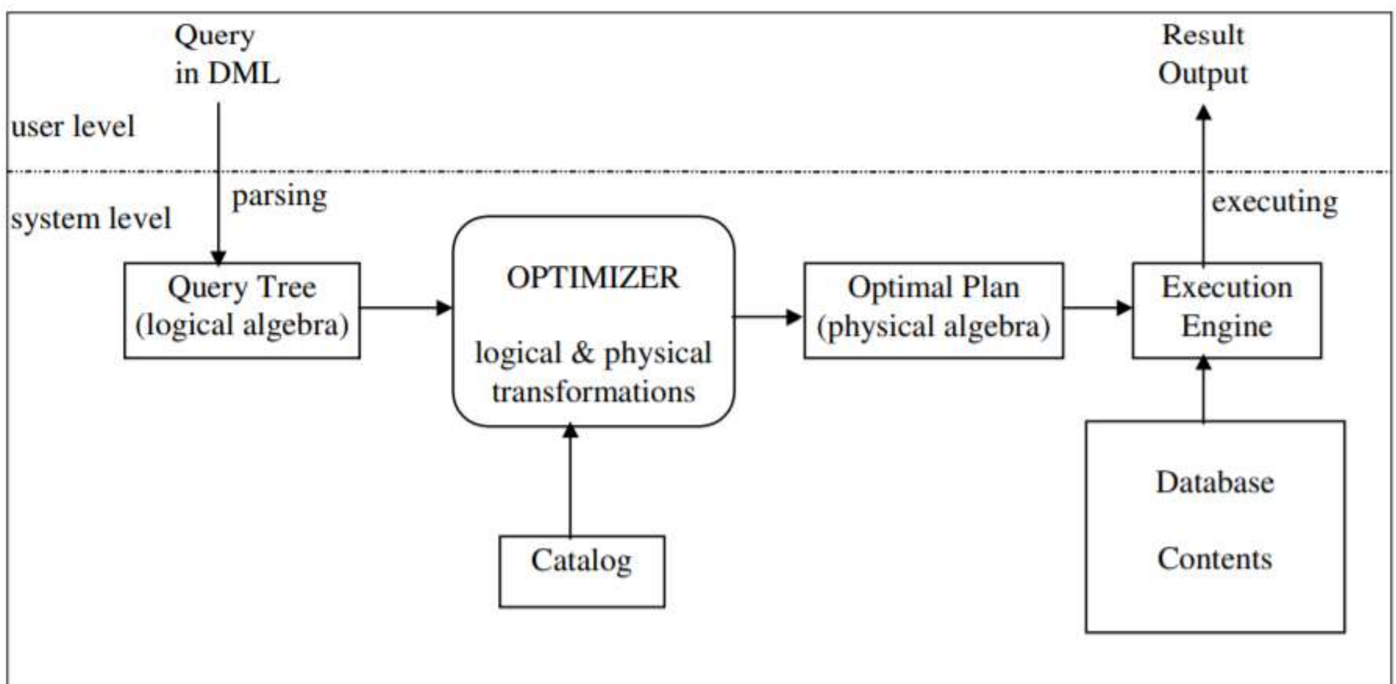


Figure 1. Query Processing

优化过程中，优化器需要搜索最优逻辑、物理计划，需要评估每种计划的代价；由此可以将优化器分解为三**大部分**。（事实上，每个都是一门研究方向。）

- **搜索计划空间**（Plan Enumeration）：Volcano/Cascades 采用**动态规划**（Dynamic Programming），继承自[9]System-R；搜索**自顶向下**。搜索需要在短暂时间内探索大量逻辑、物理计划，并评估代价。
- **代价模型**：如何为一个物理计划（Physical Operator 树）确定代价，如何为每个 Physical Operator 确定代价。Logical Operator 需展开为 Physical Operator 后才能确定代价，但能提前估算更好。
- **统计信息**（Statistics）：代价模型需要统计信息作为输入，例如 Selectivity、Cardinality。经过 Operator 输出后的"表"，例如 Join 结果，也需要导出统计信息（Statistics Derivation），为下一步计算代价。

基本概念 – Operator

Operator、Logical Operator、Physical Operator 已在"基本名词"一节讲过，不再敷述。

Volcano/Cascades 中，它们抽象了 SQL 的基本操作单元，接口基本是输入行-输出行。存储引擎只需按照接口实现 Physical Operator，即可执行优化器的输出的执行计划（Operator 树）。

Volcano/Cascades 通过 Operator 的抽象，划分了优化器和存储引擎的界限，也让扩展变得简单。

基本概念 – Pattern/Rule

搜索过程驱动优化器工作，不断生成等价的 Operator 树（逻辑计划、物理计划），扩展搜索空间，寻找最优。如何为其建立精炼的抽象概念？Volcano/Cascades 的答案是 **Pattern** 和 **Rule**

- **Pattern**：当 Operator 树的某个局部形态匹配 Pattern 时，可以对其应用 Rule 来转换树的形态；例下图[10P62]。
- **Rule**：如何转换 Operator 树，即将其匹配的节点替换（Substitute）为新形态。从而生成新的、等价的 Operator 树。

Pattern: (L(1) join L(2)) join L(3)

Substitute: L(1) join (L(2) join L(3))

逻辑优化和物理优化都需不断应用 Pattern/Rule。在逻辑优化阶段

- **Transformation Rule**：逻辑优化应用 Pattern/Rule 生成新的 Operator 树（逻辑计划）以供探索；例如根据 Rule 生成不同 Join 顺序的多个逻辑计划。此时的 Rule 称作 Transformation Rule（对应 Implementation Rule）。
- **Normalization**：逻辑优化中有些总是更优的转换，例如谓词下推、去掉不需要的 Select 字段（Cropping）。这些转换（Transform）往往在优化开始阶段执行；也可由 Pattern/Rule 定义。CockroachDB 称其为 Normalization[5]。

在物理优化阶段

- **Implementation Rule**：将 Logical Operator 转换为对应的（多种）Physical Operator 的 Rule 称作 Implementation Rule。该过程生成新的（多个）Operator 树，逐步优化，最终生成完全由 Physical Operator 组成的物理计划。

至此，**优化器搜索过程被抽象为不断匹配 Pattern 然后应用 Rule 转换**，搜索空间递归展开，应用代价模型择优。开发者只需添加新的 Pattern 和 Rule 即可扩展优化器（如继承抽象类、添加 Optgen[37]文件）。

基本概念 – Memo

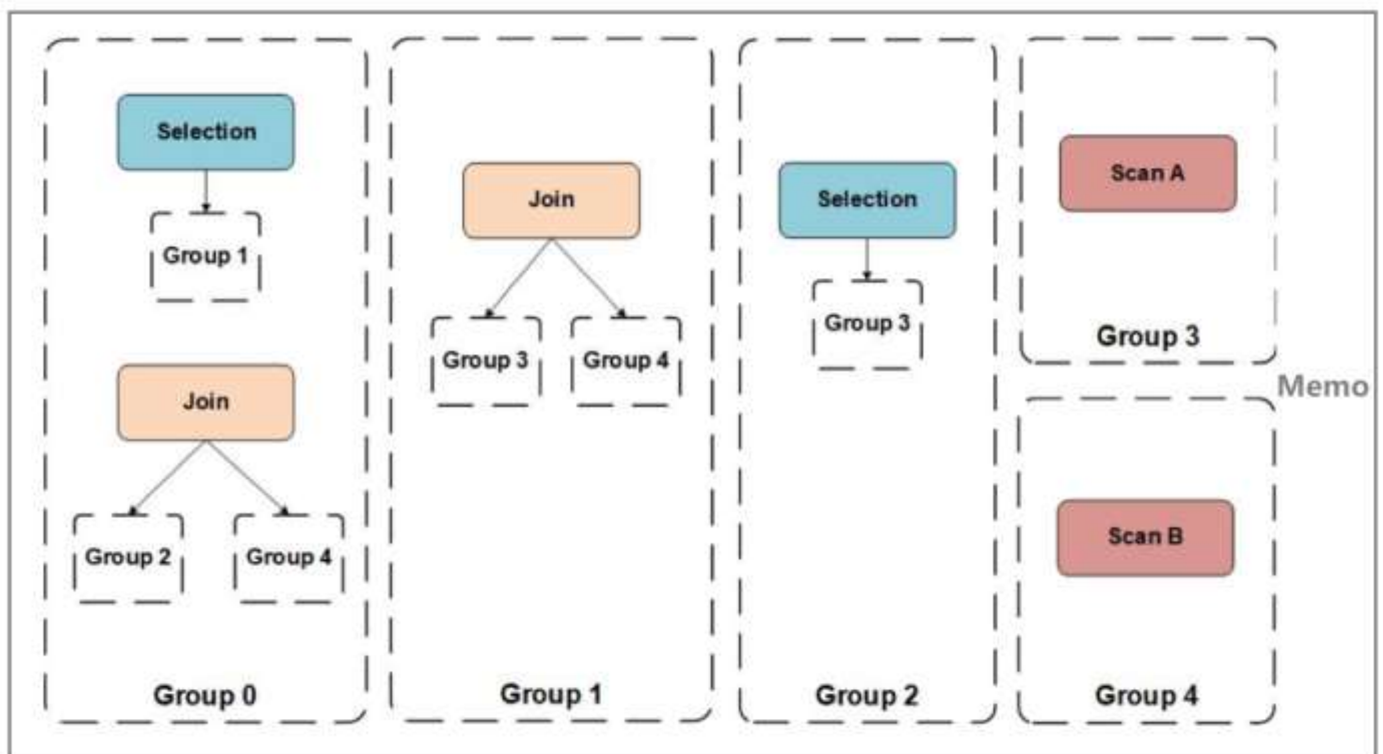
上文将搜索过程抽象为 Pattern/Rule；之后还有一大特性需要抽象，那就是**"等价"**。如何表达它呢？Volcano/Cascades 使用 Group Expression

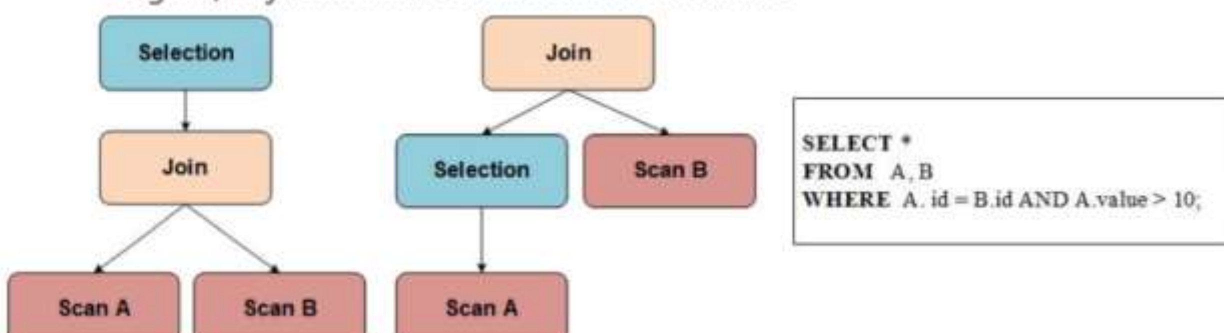
- **Expression**：关系代数表达式，对应 Operator 树中一个节点。优化器需搜索 Expression 的不同等价形式，选择代价最优
- **Group Expression**：一个 Operator 有多种 Logical Operator、Physical Operator 供选择，它们互相等价，编为一组，称作 Group Expression（其实称作 Expression Group 更易理解）。Group Expression 也可看作一个 Expression；Operator 树中，节点实际指向的是 Group Expression，这样就将可选的等价范围表达在树中了。

动态规划搜索中，需要维护**共享数据结构**。很明显应该基于 Group Expression 构造它，Volcano/Cascades 中称其为 Memo

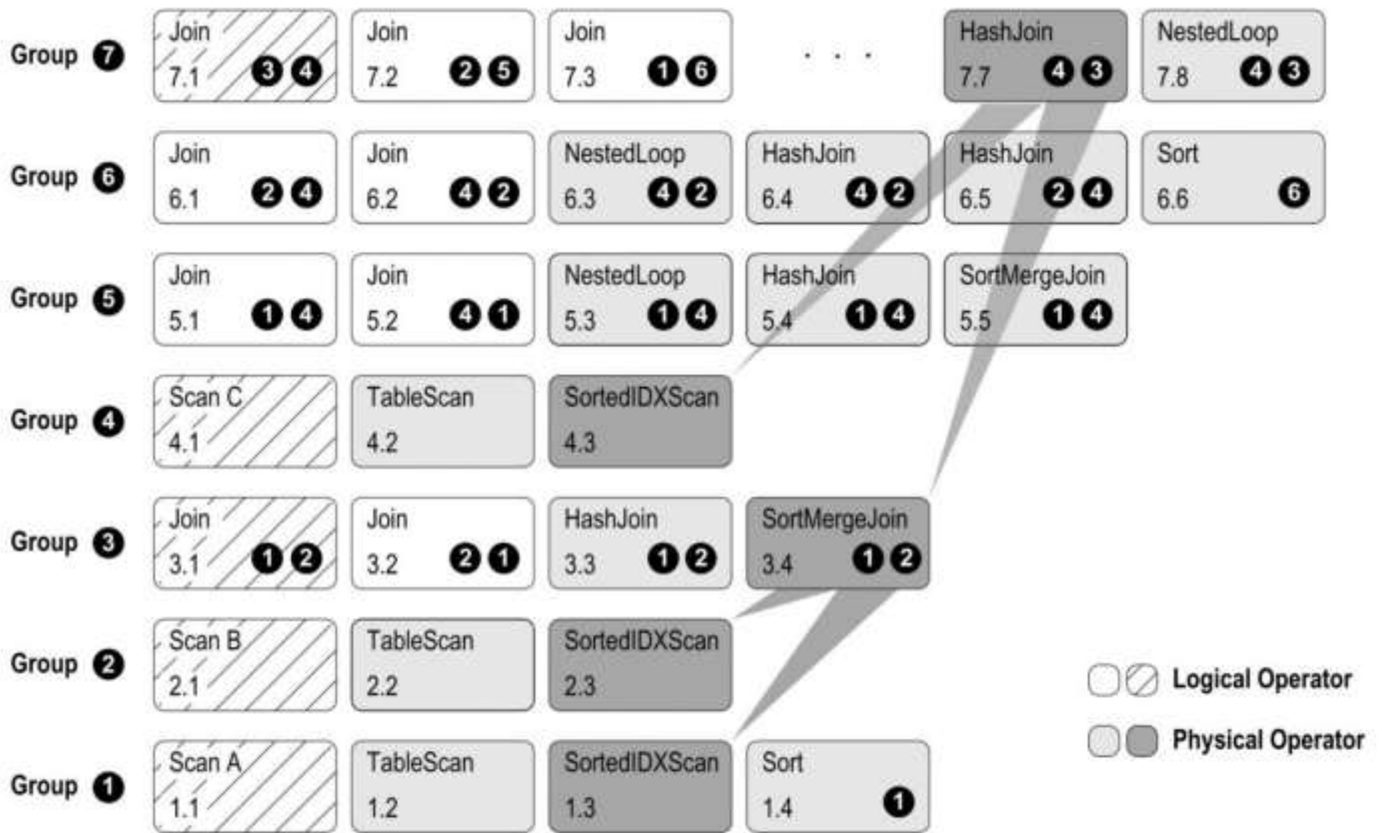
- **Memo** 如何记录**可探索**的计划空间？Memo 中各 Group 指 Group Expression，保存等价的 Expression，可以是 Logical 或 Physical Operator。Group 0 是表达式树的顶层，父 Expression 的子指针指向后续 Group Expression。从 Group 0 树遍历即可穷举计划空间。
- **Memo** 如何记录**被选中**的逻辑/物理计划？父 Expression 的子指针指向具体 Expression 即可，而不是 Group Expression。动态搜索可以从子树开始，逐步构建最优结构，然后递推至父。

Memo 例子 1[2]





Memo 例子 2[1]



基本概念 – 任务调度

优化器搜索过程本质上是递归地动态规划，如何将其实现的易于扩展、能够多核调度呢？通常是实现为任务（Task/Job）调度框架。

下图来自 Columbia[10P15]优化器。任务在执行过程中会创建新任务，**推入栈**中；执行器从栈中反复抽取任务执行，直到耗尽。任务有多种类型，如优化 Group Expression、应用 Rule 转换、计算代价。

```
optimize()
{
        // start optimization with top group
        PTasks.push ( new O_GROUP ( TopGroup ) );

        // main loop of optimization
        // while there are tasks undone, do one
        while (! PTasks.empty ())
        {

                TASK * NextTask = PTasks.pop ();        // get the next task
                NextTask -> perform ();                 // perform it

        }

        // optimization completed, copy out the best plan
        Ssp.CopyOut() ;
}
```

Figure 16. Main Loop of Optimization in Columbia

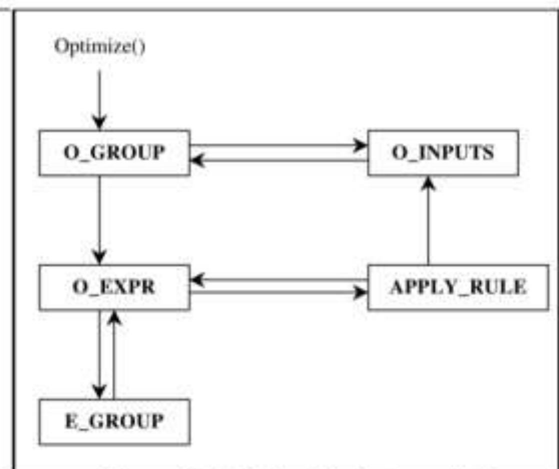


Figure 17. Relationship between Tasks

下图来自 Orca[12]优化器，任务划分有所不用，但其展示了任务之间的**依赖关系**。递归中单一的 Stack 关系被更丰富准确的图关系所表达。调度器依照任务依赖图，用 Task Pipeline 方式在多核并行运行。

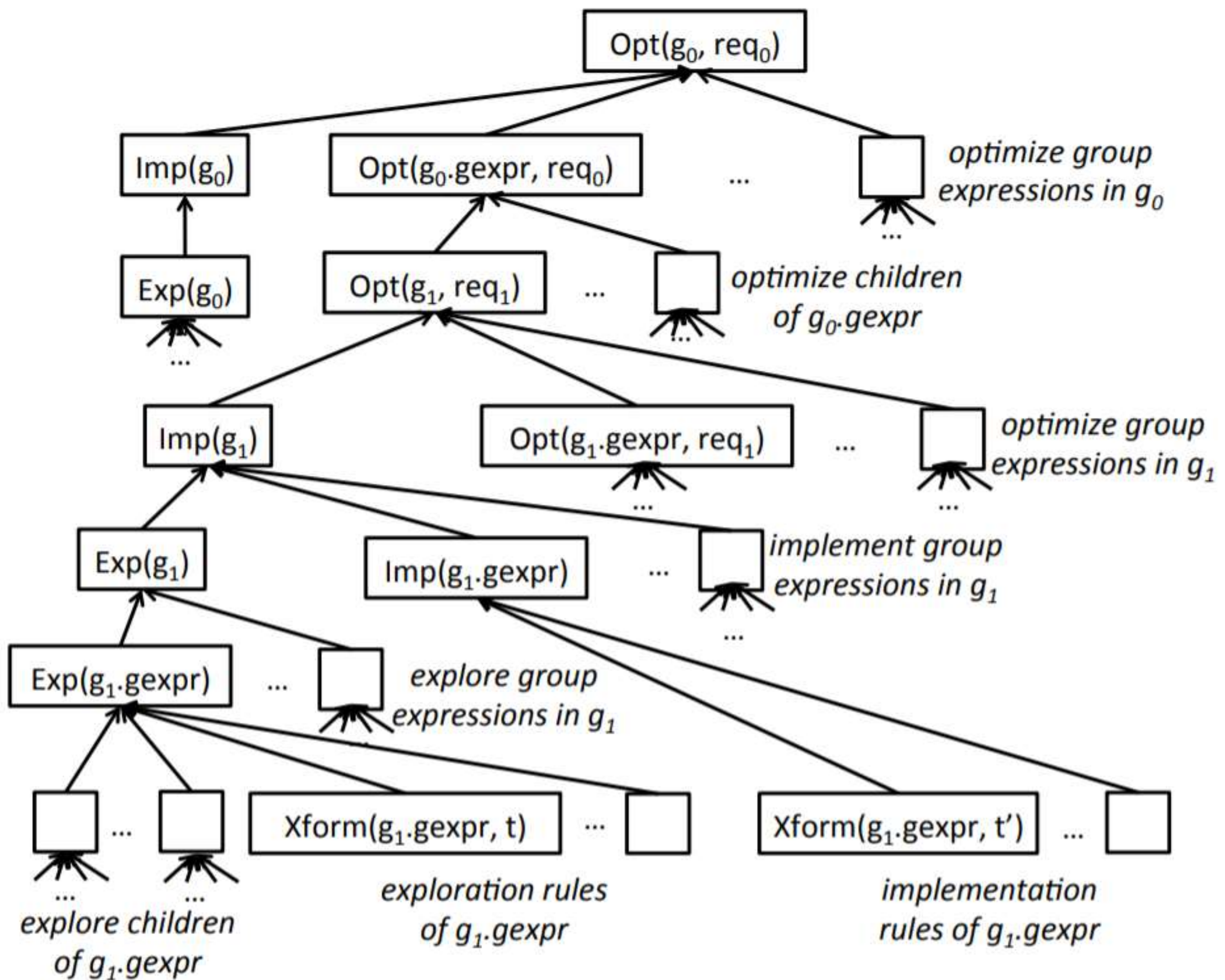


Figure 8: Optimization jobs dependency graph

Volcano 与 Cascades 的区别

Cascades 由同一作者在 Volcano 后提出，相对 Volcano 比较关键的改进有

- Cascades **合并**了逻辑优化和物理优化。Volcano 则分为逻辑优化阶段和物理优化阶段，在逻辑优化阶段需穷举逻辑计划，但后期未必用得上，浪费了搜索资源。以此类推，Cascades 也不再（严格）区分 Logical Operator 和 Physical Operator，Transform Rule 和 Implementation Rule
- Cascades 提出了 **Memo** 数据结构，详见上文。Memo 此后被广泛采用[41]，成为优化器的核心之一。

搜索计划空间（Plan Enumeration）

上文已基本讲解搜索过程， 下面我们更加深入，并且带出 Volcano/Cascades 的另外几个概念：Interesting Order、Promise 、Logical Property、Physical Property、 (Property) Enforcer。

局部最优问题

动态规划的基本假设是子树最优导出父最优（最优子结构）， 但对于查询优化这是不正确的。例如，子树采用 Hash Join 也许更快，但结果无序，父需额外排序；Sort-merge Join 更慢，但结果有序；两者并无确定优劣。

首先，如何表达这种差异呢？

- **Physical Property**：Operator 返回的结果带有的属性，最典型的例子是排序与否；还例如结果是否已计算 Hash；结果在单服务器上，还是分散在多服务器（需要额外 GatherMerge）。

在搜索中，稳妥的做法是子树对每种不同的 Physical Property 都返回一个最优计划，供父节点选择。父节点也可指定自己关注的 Physical Property，然后让子树依此开始搜索

- **Interesting Order**：（也许叫 Interesting Property 更合适，因为多数 Interesting 的都是排序属性。）由父节点传给子节点，要求其返回满足特定 Physical Property 的最优计划。
- 另一个能够解释"Interesting Order"名称的是 **Join Order**。在多表互 Join 时，表的 Join 顺序（Join Order）排列组合数量巨大，且以数量级级别[14]影响查询性能；父节点需选择哪些 Join Order 应被进一步探索（Join Enumeration）。它们成了"Interesting Order"。

由此，动态规划局部最优问题得到解决。相比标准算法只保留一个局部最优结果，查询优化器将所有可能达成全局最优的局部最优列为候选，依次保留和探索。

关于 Property 和 Enforcer

解释了 Physical Property，那么 Logical Property 是做什么？

- **Logical Property**：主要出自 Volcano 论文，它是从逻辑优化阶段的关系代数表达式中提取的属性，主要包括 Operator 输入输出的 Schema（如有哪些列）、行数等统计信息。

什么是 Property Enforcer 呢？Operator 树中，父节点要求的 Physical Property 可能在紧邻的子节点上是缺失的。例如父 Operator 要求结果排序，但子 Operator 给出无序的结果。此时需要"Enforce the missing property"

- **Property Enforcer**：（也简称 Enforcer）上例中，Property Enforcer 会在父子节点间插入一个新的节点 Sort Operator，使结果排序。即补上缺失的 Physical Property。

更复杂的例子[12]如下图。(b)要求输出 Physical Property {Singleton,<T1.a>}，即结果位于单服务器，且对 T1 表的 a 字段排序。(c)中 Property Enforcer 补上了缺失的 Physical Property，左图选择先尝试添加排序<T1.a>，而右图选择先尝试添加单服务器 Singleton。

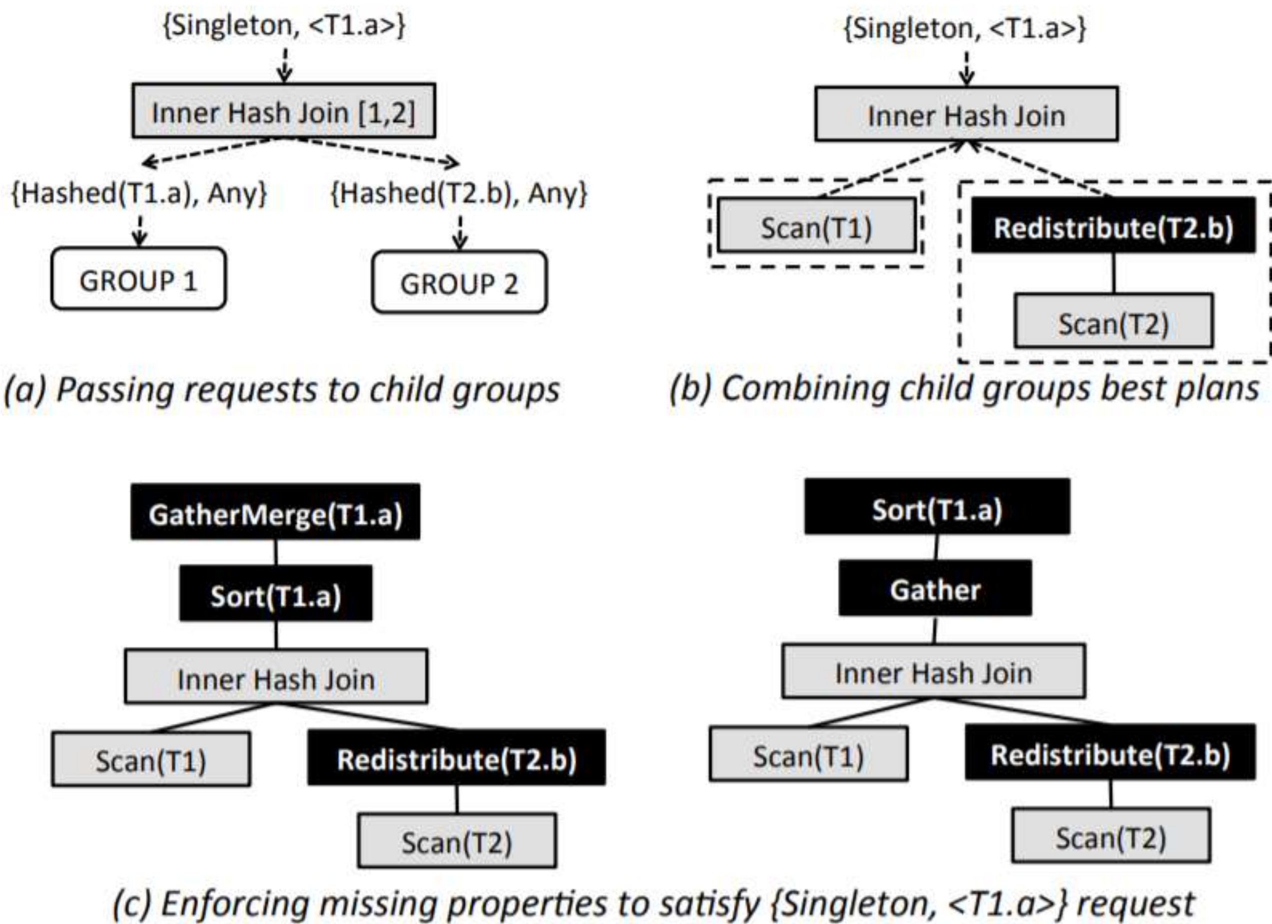


Figure 7: Generating InnerHashJoin plan alternatives

搜索优先级

优化器搜索时间有限，而分支众多，不一定能全部穷举；此时需要优先选择“看上去”更优秀的分支。如何表达这一抽象呢？Volcano/Cascades 中将其表达为 Promise

- **Promise**：前文中可见，发现一个搜索“分支”对应一个 Rule 的转换。每个 Rule 对外提供一个 Promise 值（正整数），越高表示该分支更有可能找到最有计划，而搜索任务调度器会优先选择。

开发者通过设定 Promise 值，可以引导优化器偏向预设的搜索空间。而 Promise 为零或负数时，则可禁止 Rule 被执行。

Promise 值计算通常是启发式的（基于规则的），例如[12]优先选择更少 Join 条件的 Inner Join，以规避更多 Join 条件带来的更大代价估算偏差。

除 Promise 外，Orca[12]还实现了 Multi-Stage Optimization。Rule 被分入不同 Stage，优先执行资源消耗少的。各 Stage 可配置不同的退出条件。

避免重复搜索

动态规划中，同一个表达式可能被重复地发现并搜索，相同表达式可能被不同的 Rule 转换出。如何去重呢？

- **Duplicate Expression Detection**：通常采用 Hash 方法[10P48]，表达式可计算出其对应的 Hash 值，放入全局缓存中。去重功能甚至可以内置（Build-in）到 Memo 中。

可看到，上面引入了表达式缓存。SQL 计划的优化结果能够被缓存，Operator 树的子树也能被缓存。预编译查询往往能够更好利用缓存。

剪枝（Pruning）
动态规划搜索常常可以利用上下界剪枝，去掉不必要的搜索分支；查询优化器也不例外。如果已经搜索了一些候选计划，那么代价比它们更高的计划就可以跳过

- **Branch-And-Bound Pruning**：已搜索完成的物理计划的代价最小值成为 Cost Upper Bound。当新的搜索分支的代价高于它时，不需继续搜索。初始 Cost Upper Bound 可由优化器根据启发式规则估算。

此外，Columbia 还实现了更多剪枝[10P83]算法，如 Global Epsilon Pruning。

搜索退出条件
最后，动态规划搜索应该何时退出呢？典型[12]方法有

- **搜索退出条件**：1）低于预设代价的物理计划被找到；或 2）超时；或 3）搜索空间穷举完毕。

Join Order Enumeration
从上文可以看出，搜索过程中，父节点需提供子树应探索的不同 Join Order。尤其是 OLAP 数据库，Join 参与表数量通常较多，可选 Join Order 排列组合成几何数量增长。而 How Good Optimizers 论文[14]指出，Join Order 能影响 SQL 执行速度达数量级级别。

Join Order 并没有完美的解决方案，常见方法如 PostgreSQL[43]所用

- 当参与 Join 的表数量少于上限时（配置参数），**遍历**所有排列组合。否则，基于启发式规则选择一些 Join Order，例如**随机**生成。此外，允许用户 Explicit 强制设定优化器采用的 Join Order。

Oracle[45]数据库更详细地设计了如何用**启发式**规则猜测更优的 Join Order

- 如果表更小，对 Join 条件有索引，需扫描的行更少；那么这个表更可能被排列到 Join Order 靠前。

MemSQL[13]针对 OLAP **雪花表**（Snowflake Schema），设计了特殊的启发式规则优化 Join Order

- 依靠雪花表的图关系（不需代价信息），区分 Satellite 节点（通常为 Fact 表，自带 Filter 条件），和 Seed 节点（通常为 Dimension 表）。调整 Join Order 以形成 Bushy Join，方便谓词下推。

搜索算法的详细例子

Columbia 的详细例子
这个算法来自与 Columbia[10]论文，详细讲述了 Cascades 优化器的实现（以及效率改进）。下图为其任务划分，以及调用、创建关系

- O_GROUP：优化一个 Group Expression
- E_GROUP：展开（Expand）一个 Group Expression，优化其中每个表达式
- O_EXPR：优化一个表达式
- APPLY_RULES：匹配（Bind）Pattern 并应用 Rule 转换，新表达式被放入任务栈继续优化
- O_INPUTS：计算表达式的代价，并决定剪枝

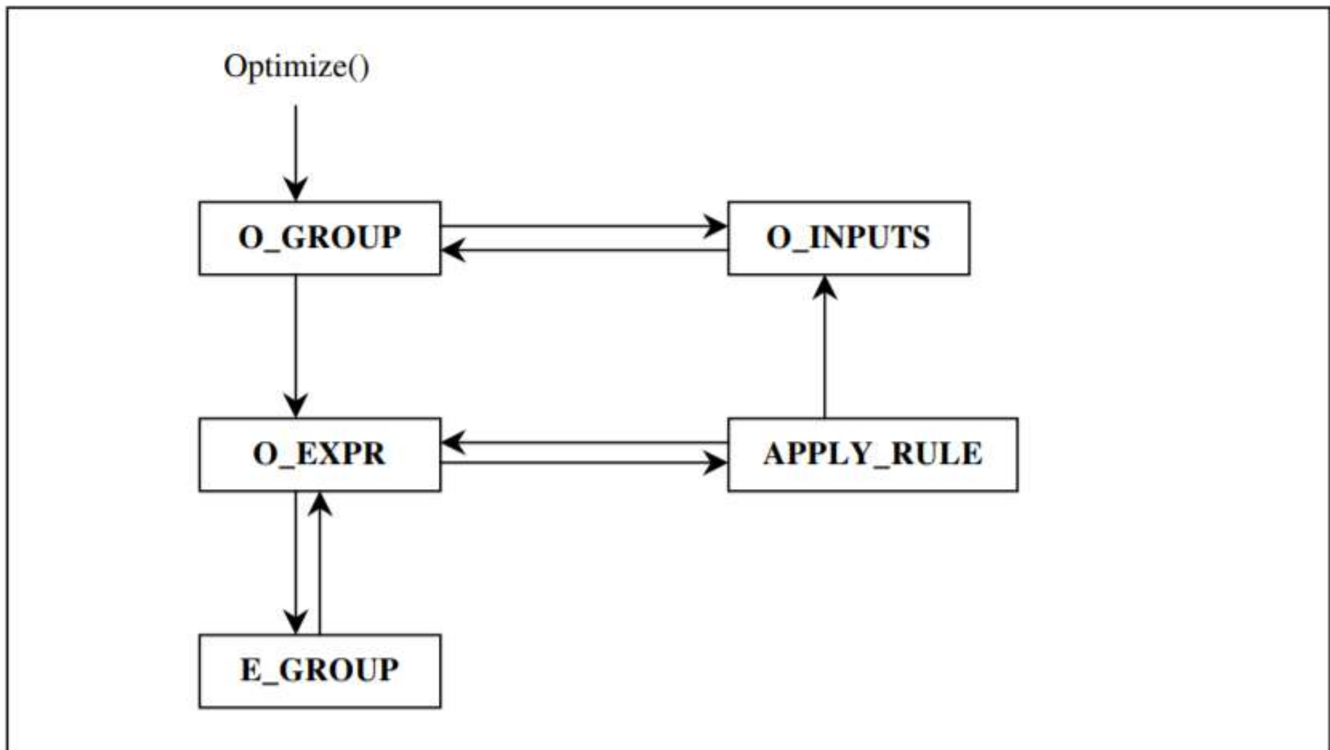


Figure 17. Relationship between Tasks

可以看见，优化器的算法驱动是循环调用任务栈

```
optimize()
{
        // start optimization with top group
        PTasks.push ( new O_GROUP ( TopGroup ) );

        // main loop of optimization
        // while there are tasks undone, do one
        while (! PTasks.empty ())
        {

                TASK * NextTask = PTasks.pop ();        // get the next task
                NextTask -> perform ();                 // perform it

        }

        // optimization completed, copy out the best plan
        Ssp.CopyOut() ;
}
```

Figure 16. Main Loop of Optimization in Columbia

优化初始 Operator 树，作为一个 O_GROUP 被推入任务栈。

```
//find the best plan for a group with certain context
O_GROUP::perform( context )
{

        If ( lower bound of the group greater than upper bound in the context)
                return;                              // impossible goal

        If ( there is a winner for the context )
                Return;                              // done, no further optimization needed

        // else, more search needed

        // optimize all the logical mexprs with the same context
        For ( each logical log_mexpr in the group )
                PTasks.push (new O_EXPR( log_mexpr, context ) );

        // cost all the physical mexprs with the same context
        For ( each physical phys_mexpr in the group )
                PTasks.push ( new O_INPUTS( phys_mexpr , context ) ) ;

}
Note: Since the tasks are pushed into a stack, O_INPUTS tasks are actually scheduled earlier than
O_EXPR tasks. It is desired because a winner may be produced earlier.
```

Figure 18. Algorithm for O_GROUP

E_GROUP 展开 Group Expression

```
// derive all logical multi-expression for matching a pattern
E_GROUP::perform(context)
{
        If ( the group has been explored before)
                Return;

        // else,  the group has not yet been explored
        for ( each log_mexpr in the group )
                PTasks.push (new O_EXPR( log_mexpr, context, exploring ) );

        Mark the group explored;
}
```

Figure 19. Algorithm for E_GROUP

O_EXPR 优化指定的 Expression。Logical Operator 在此通过应用 Implementation Rule，被转换成 Physical Operator。

```
// optimize or explore a multi-expression, firing all appropriate rules.
O_EXPR::perform( mexpr, context , exploring )
{
        // Identify valid and promising rules
        For (each rule in the rule set)
        {
                // check rule bit in mexpr
                if ( rule has been fired for mexpr )  continue;

                // only fire transformation rules for exploring
                if (exploring && rule is implementation rule ) continue;

                // check top operator and promise
                if (top_match(rule, mexpr) && promise(rule,context) > 0 )
                        store the rule with the promise;
        }

        sort the rules in order of promises;

        // fire the rules in order
        for ( each rule in order of promise)
        {
                // apply the rule
                PTasks.push ( new APPLY_RULE ( rule, mexpr, context, exploring ) );

                // explore the input group if necessary
                for (each input of the rule pattern )
                {
                        if ( arity of input > 0 )
                                PTasks.push ( new E_GROUP( input grp_no, context) );

                }
        }
}
```

Figure 20. Algorithm for O_EXPR

Columbia 为增加 Pattern 匹配速度做了一些优化，例如首先测试 Pattern 的根节点是否匹配，否则跳过。

```
// apply a transformation or implementation rule
APPLY_RULE::perform( mexpr, rule, context, exploring )
{
        // check rule bit in mexpr
        if ( rule has been fired for mexpr )  return;

        for (each binding for the mexpr and rule)
        {
                before = binding->extract_expr();              // get the binding expression

                if ( rule->condition(before) not satisfied )   continue;  // check condition

                after = rule->next_substitute(expr );          // get the substitute from the rule

                new_mexpr = Ssp->CopyIn(after);                // include the substitute into SSP

                // further transformations to optimize new expr
                if ( new_mexpr is logical )
                        PTasks.push (new O_EXPR (new_mexpr, context, exploring));

                // calculate the cost of the new physical mexpr
                if ( new_mexpr is physical )
                        PTasks.push (new O_INPUTS (new_mexpr, context));

        }

        mexpr->set_rule_bit(rule);                     // mark the rule has been fired

}
```

Figure 21. Algorithm for APPLY_RULE

O_INPUT 会把自己重复推到栈上，自我重入，以迭代计算整个物理计划的代价。

```
//On the first (and no other) execution, the code initializes O_INPUTS member InputCost.
For each input group IG
        If (Starburst case)  InputCost is zero;
        Determine property required of search in IG;
        If (no such property)  terminate this task.;
        Get Winner for IG with that property;
        If (the Winner from IG is a Full winner)  InputCost[IG] = cost of that winner;
        else if (!CuCardPruning)  InputCost[IG] = 0      //Group Pruning case
            else if (no Winner)  InputCost[IG] = GLB    //remainder is Lower Bound Pruning case
                else        // Winner has a null plan, find a lower bound for IG
                            InputCost[IG] = max(cost of winner, IG Lower Bound)
EndFor  // initialize InputCost

//The rest of the code should be executed on every execution of this method.
If (Pruning && CostSoFar >= upper bound) terminate this task.   // Note1: group pruning applied

//Calculate cost of remaining inputs
For each remaining (from InputNo to arity) input group IG;
  Probe IG to see if there is a winner;
  If (there is a Full Winner in IG)
        store its cost in InputCost;
        if (Pruning && CostSoFar exceeds G's context's upper bound) terminate this task;
  else If (we did not just return from O_GROUP on IG)
                //optimize this input; seek a winner for it
                push this task;
                push O_GROUP for IG;
                return;
      else   // we just returned from O_GROUP on IG, this is an impossible plan
          if(There is a winner in IG with a null plan)
                If appropriate, update null-plan winner in IG;
                terminate this task;
          else  // There is no winner in IG
                Create a new null-plan winner in IG;
                terminate this task;
EndFor //calculate the cost of remaining inputs

//Now all inputs have been optimized
if (arity==0 and required property can not be satisfied)  terminate this task;
if (CostSoFar >= than G's context's upper bound)  terminate this task;

//Now we know current expression satisfies current context
if (GlobepsPruning && CostSoFar <= GLOBAL_EPS)       // Note2: global epsilon pruning applied
        Make current expression a winner for G;
        mark the current context as done;
        terminate this task;

if (either there is no winner in G or CostSoFar is cheaper than the cost of the winner in G)
        //so the expression being optimized is a new winner
        Make the expression being optimized a  new winner
        update the upperbound of the current context
        return;
```

Figure 22 Pseudo-code of O_INPUTS::perform()

Memo 的详细例子

Orca[12]论文中的 Memo 图非常有益说明，并且是分布式查询优化。图中左中右分别是

- **Group Hash Table**：Orca 中的优化任务称作 Request，记录于 Group Hash Table 中，Request 去重通过 Hash 完成。图中 Group Hash Table 左中右字段分别是 Request ID，Request 输出的内容和其 Physical Property，代价最优的表达式（指向中图 Memo）。
- **Memo**：如前文所述。一个 Group 中汇聚了输出内容相同，但 Physical Property 不同的表达式。每个表达式的子指针指向 Group Hash Table 中的 Request ID。
- **Extracted Final Plan**：根据图中 Group Hash Table 和 Memo 和指针关系，可以构建出来的代价最优的物理计划

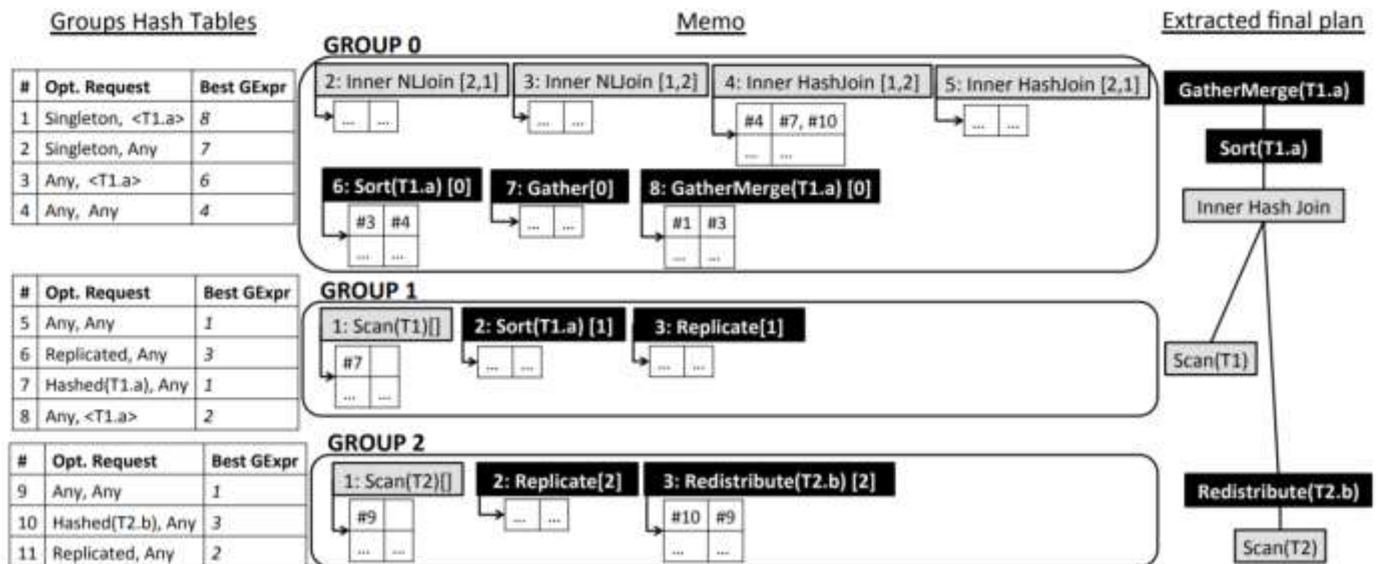


Figure 6: Processing optimization requests in the Memo

进一步解释图中内容；下面跟随指针关系，如下各步可以组装出 Extracted Final Plan

1. 优化器首先输入的是 Request#1，要求查询表`T1 join T2 by T1.a == T2.b`，结果位于单个服务器，且按 T1.a 排序
2. Request#1 的 Best GExpr 指向 Memo 中的 8: GatherMerge(T1.a)，后者指向 Request#3。
3. Request#3 仍输出`T1 join T2 by T1.a == T2.b`，但不要求结果位于单个服务器；这是因为其父 GatherMerge 汇聚结果。
4. Request#3 的 Best GExpr 指向 6: Sort(T1.a)，后者指向 Request#4
5. Request#4 仍输出`T1 join T2 by T1.a == T2.b`，但不要求排序；这是因为其父 Sort 了结果
6. Request#4 的 Best GExpr 指向 4: Inner HashJoin，后者指向 Request#7 和 Request#10
7. Request#7 位于 GROUP 1，它扫面 T1，并且为 T1.a 计算 Hash。它的 Best GExpr 指向 1: Scan(T1)
8. Request#10 位于 GROUP 2，从指针关系可以看到其首先 Scan(T2)，然后 Redistributed(T2.b)。Redistribute 指按照 T2.b 的 Hash 值分散数据到对应服务器上，由此可以 Join 于 T1.a。

由上面也可发现，Memo 结构常常设计得节省内存（Memory Compact），以便有空间存放更多的表达式以供搜索。

代价模型（Cost Model）

简单直接地说，代价模型是 Operator 实现接口中的 GetCost[39]函数，返回数值表示代价高低；例如下图[38]。最后 Operator 树遍历求和整个 SQL 表达式的代价。

- `IndexMergeType` = 1

  IO Cost = (totalRowCount + mergedRowCount) * scanFactor

  Network Cost = (totalRowCount + mergedRowCount) * networkFactor

  Cpu Memory Cost = totalRowCount * cpuFactor + totalRowCount * memoryFactor

基础 Cost Variable

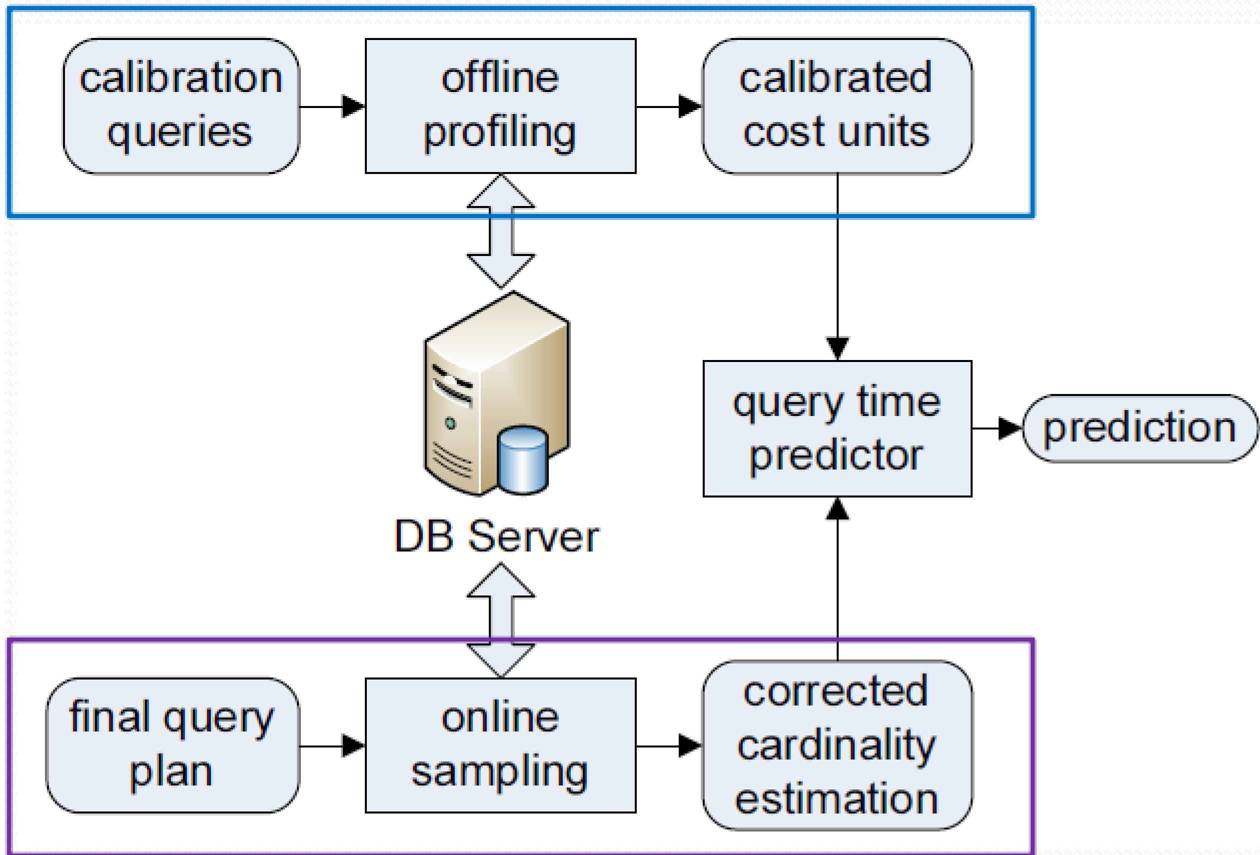
代价模型的底层通常来自 CPU、内存、网络/磁盘 IO 等维度，在模型中加权求和得到最终分数。这些权重——即“单位代价”——如何取值是模型的基础。但常见方法仅仅是在配置文件中指定（如下图 TiDB[40]、Columbia[10P110]、PostgreSQL[42]）；PostgreSQL 称它们为 **Cost Variables[42]**。

```
// tidb_opt_cpu_factor is the CPU cost of processing one expression for one row.
TiDBOptCPUFactor = "tidb_opt_cpu_factor"
// tidb_opt_copcpu_factor is the CPU cost of processing one expression for one row in coprocessor.
TiDBOptCopCPUFactor = "tidb_opt_copcpu_factor"
// tidb_opt_tiflash_concurrency_factor is concurrency number of tiflash computation.
TiDBOptTiFlashConcurrencyFactor = "tidb_opt_tiflash_concurrency_factor"
// tidb_opt_network_factor is the network cost of transferring 1 byte data.
TiDBOptNetworkFactor = "tidb_opt_network_factor"
// tidb_opt_scan_factor is the IO cost of scanning 1 byte data on TiKV.
TiDBOptScanFactor = "tidb_opt_scan_factor"
// tidb_opt_desc_factor is the IO cost of scanning 1 byte data on TiKV in desc order.
TiDBOptDescScanFactor = "tidb_opt_desc_factor"
// tidb_opt_seek_factor is the IO cost of seeking the start value in a range on TiKV or TiFlash.
TiDBOptSeekFactor = "tidb_opt_seek_factor"
// tidb_opt_memory_factor is the memory cost of storing one tuple.
TiDBOptMemoryFactor = "tidb_opt_memory_factor"
// tidb_opt_disk_factor is the IO cost of reading/writing one byte to temporary disk.
TiDBOptDiskFactor = "tidb_opt_disk_factor"
// tidb_opt_concurrency_factor is the CPU cost of additional one goroutine.
TiDBOptConcurrencyFactor = "tidb_opt_concurrency_factor"
```

（对，一切困难问题都可以代理给配置文件，或用接口推给具体 Driver 实现 ...）

> **Note**
>
> Unfortunately, there is no well-defined method for determining ideal values for the cost variables. They are best treated as averages over the entire mix of queries that a particular installation will receive. This means that changing them on the basis of just a few experiments is very risky.

Cost Variable 并不容易正确配置，往往来自运行大量用户场景负载（Workload）的经验。一些方法中，它们可由在指定硬件上运行与预期相符的样本来校准（Calibrating）；例如使用机器学习[29]，结合离线训练与在线抽样（Sampling）。类似思想也出现过在 Quasar 调度器[24]中。



更复杂的代价模型

真实的代价模型会考虑更多 Cost Variable。行式（Row-oriented）与列式（Column-oriented）布局（Layout）、内存与磁盘数据库、OLTP 与 OLAP、单机与分布式，模型会有不同和侧重。例如

- 行的大小（影响扫描大小）
- Cache Miss
- 压缩与否（影响的扫描速度）
- 结果集（Result Set）写出速度
- 扫描共享（一次扫描服务多个查询）
- CPU 指令流水线（Pipeline）的利用率（IPC）
- 分布式[13]数据库带来的额外数据移动（Data Movement）、Shuffle、Hashing 代价

例如 Access Path Selection[6]论文中下图

另一条路径，……

Tree Traversal

$$TT = (1 + \lceil log_b(N) \rceil) \cdot \left(C_M + \frac{b \cdot C_A}{2} + \frac{b \cdot f_p \cdot p}{2} \right)$$

Leaves Traversal

$$s_i \cdot TL, \text{ where, } TL = \frac{N \cdot C_M}{b}$$

Data Traversal for Secondary Indexes

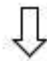
$$s_i \cdot TD_I, \text{ where, } TD_I = \frac{N \cdot (aw + ow)}{BW_I}$$

Result Writing

$$RW = s_i \cdot N \cdot \frac{rw}{BW_R}$$

Sorting the Result Set

$$SC_i = s_i \cdot N \cdot log_2(s_i \cdot N) \cdot C_A$$



$$SingleIndexProbe = TT + s_i \cdot (TL + TD_I) + RW + SC_i$$

$$ConcIndex = q \cdot TT + S_{tot} \cdot (TL + TD_I) + S_{tot} \cdot TD_R + SF \cdot CA, \text{ where}$$

$$SF = S_{tot} \cdot N \cdot log_2(S_{tot} \cdot N)$$

现代数据库还允许用户**自定义代价模型**和其依赖的统计信息（User-Defined Statistics/Selectivity/Cost[46]）。这得益于 Volcano/Cascades 框架的可扩展性，也帮助处理难以确定代价的用户定义函数（UDF）。用户还可用它们获取业务逻辑有关的数据分布信息；数据库一般提供 **SQL ANALYZE** 命令主动触发信息收集。

进一步扩展，Apache Calcite[11]允许异构数据平台以**插件**的形式接入各自的代价模型、元数据接口、Rule 等，同一 SQL 查询可联合不同数据源。

代价模型的分析和验证

How Good Optimizers 论文[14]给出了很多精彩的方法。例如，SQL 查询的真实代价是其执行时间；将代价模型的预测值与真实执行时间计算**线性相关性**，可以判断其准确程度。

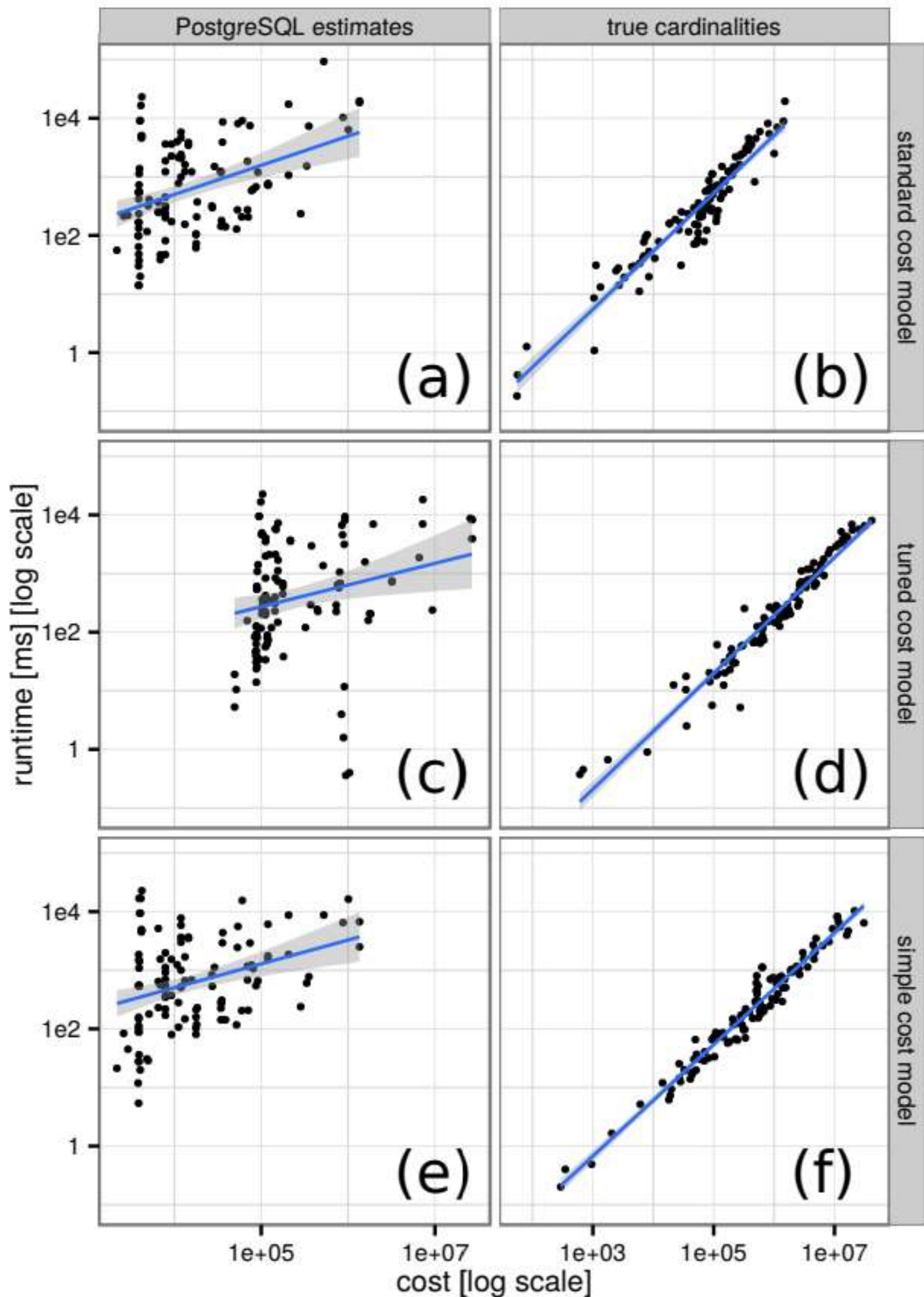


Figure 8: Predicted cost vs. runtime for different cost models

上图中(a)、(b)使用 PostgreSQL 代价模型，(c)、(d)的 PostgreSQL 额外校准了 Cost Variable，而(e)、(f)使用只考虑结果行数的简单代价模型。左列使用 PostgreSQL 估算的统计信息 Cardinality，右列使用真实准确的 Cardinality。可以发现关键结论[14]

- **准确的统计信息，即 Cardinality，远比代价模型本身重要**。即使是简单代价模型，输入准确 Cardinality，都能做出线性预测。
- 左列图左侧的大量**离群点**表示，查询优化器有时会给出估算代价很低，但实际很慢的执行计划。较低的估算代价可能来自很低但错误的 Cardinality，误导优化器选择错误的 Physical Operator，例如 Nested-loop Join。
- 对比(a)、(b)、(c)、(d)表示，**校准 Cost Variable** 有用，但远逊于准确 Cardinality 带来的提高。
- 图(c)中，查询优化器**意外**发现了耗时很短的执行计划，但却误估出很高的代价

另一个方法，下图中，该论文[14]给出了**展示 Plan Space** 的方法：遍历所有可能的执行计划，为其执行时间绘制概率密度图（PDF）。可见最优执行计划往往比中位数计划快几个数量级。

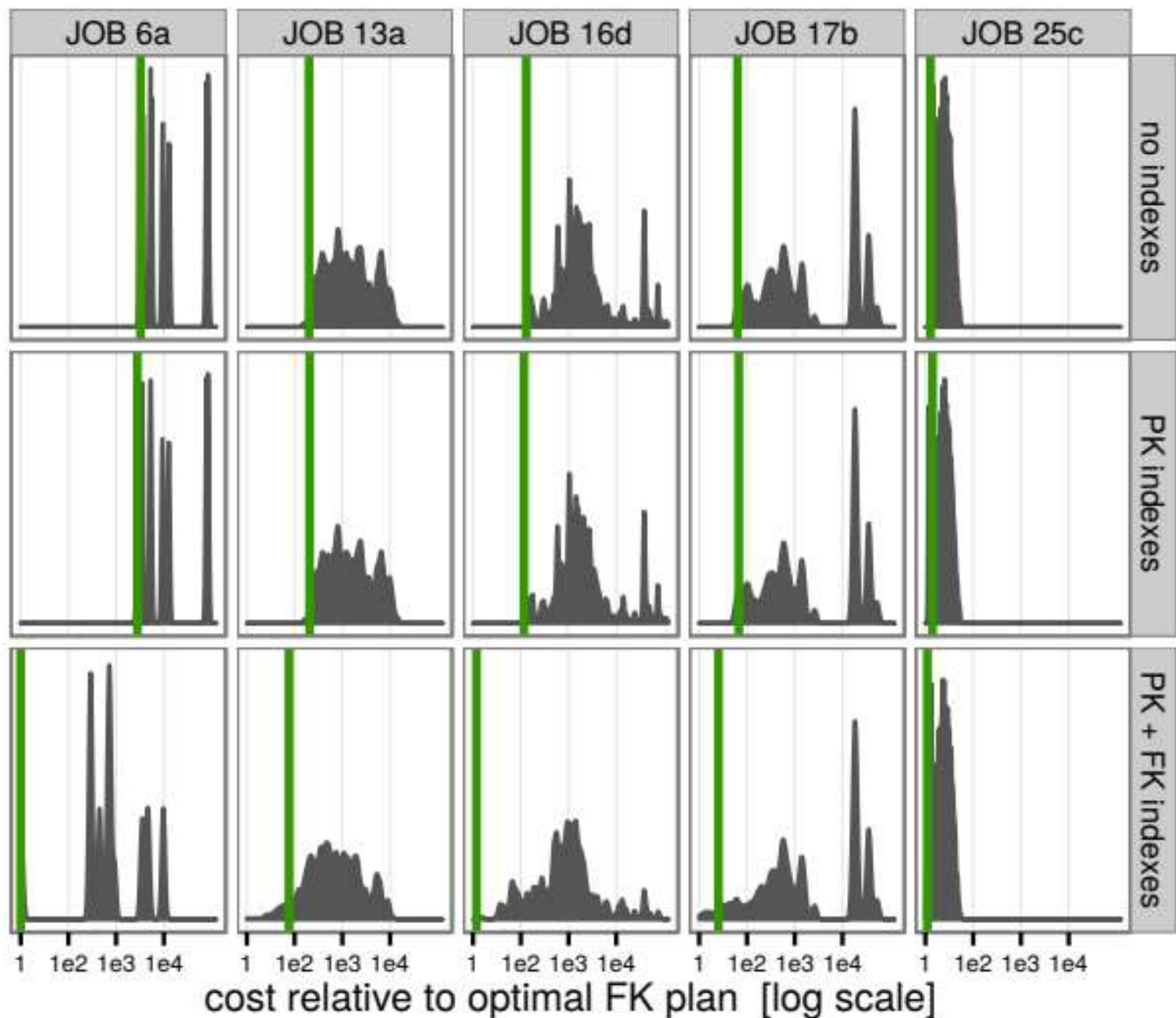


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

由此延伸，这一方法可以用来分析验证查询优化器。例如 Orca TAQO[12]

- 优化器往往只够时间搜索部分 Plan Space，被**包含**的部分包括最优执行计划吗？
- Join Order Enumeration、Promise、Interesting Order 能否将搜索过程**引导**向 Plan Space 较好的部分？（联想 SGD[31]）
- 在 Plan Space 中，代价模型估计的代价与真实执行时间，有怎样的**匹配**？最简单地，任取两个执行计划，代价模型能否正确排序？重度偏离的部分预示改进的方向

- 重点 Plan Space 可以被抽样（例如高度偏离的客户查询），组成 **Benchmark 集**，供代价模型以及优化器的回归测试和进一步优化

最终这又是一个组合[33]和空间结构[53]探索的问题。

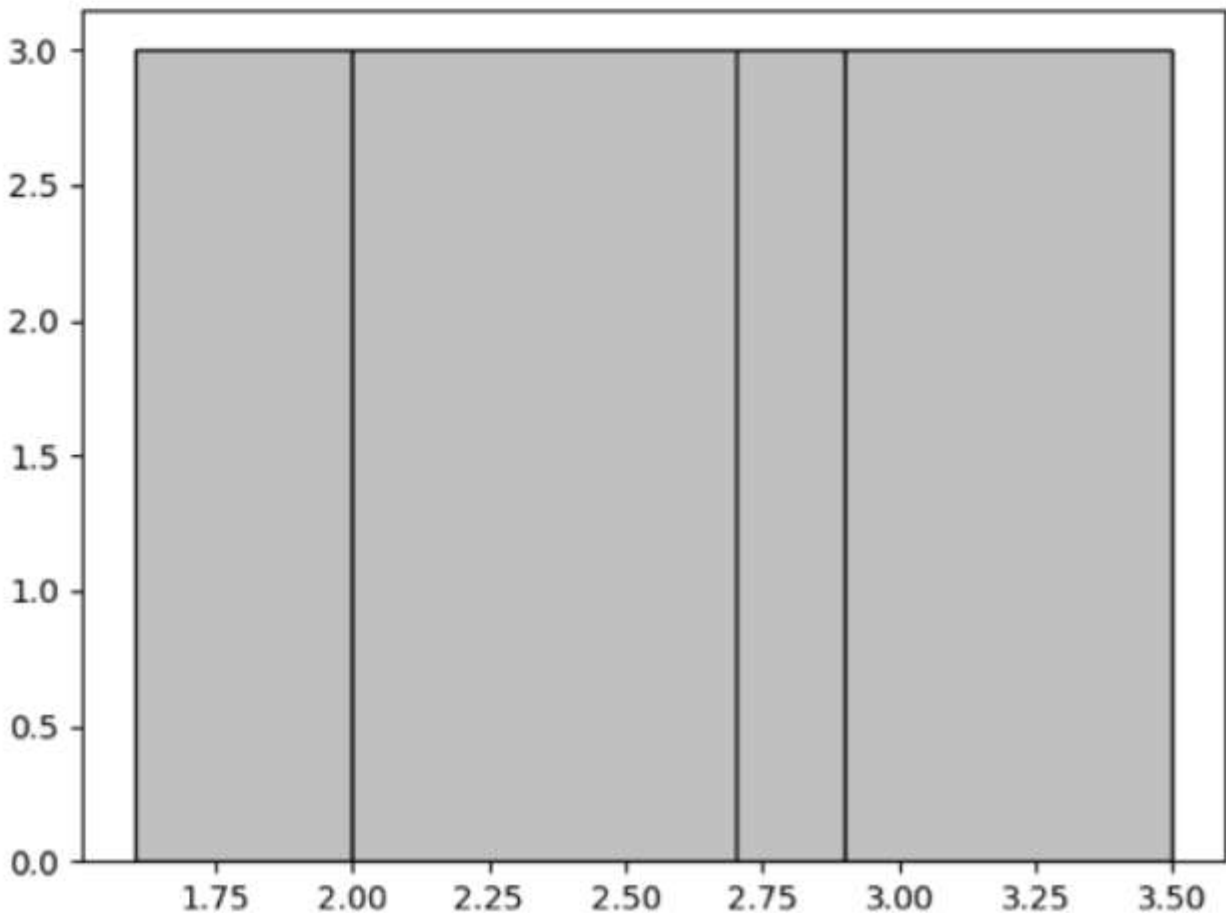
统计信息（Statistics）

数据库最主要的统计信息是 Operator 输入输出的行数，从最初的表有多少行，到中间各步 Operator，即 Selectivity 和 Cardinality（Selectivity 也可以归结为 Cardinality）。**How Good Optimizers 论文[14]指出准确的 Cardinality 远比代价模型重要**。

本文主要关注 Cardinality。此外还有 Count-Min Sketch[15]，使用类似 Bloom Filter 的方法，快速估算一个值出现的次数。（类似的"Sketching"还有 Zone map[47]，通过记录各数据块的 Min/Max 边界，帮助扫描时快速跳过。）Synopses for Massive Data[26]有更多 Sketch 方法。

柱状图（Histogram）

数据库通过柱状图记录每一列的 Cardinality，相当于离散版本概率密度图（PDF），针对列中各行取值。列值范围会被分桶（Bucket），通常**等深直方图**（Equi-Depth Histograms）有更好的效果。等深直方图指，维持每个桶中装有相等数量的行（Tuple），而不是将列值范围等分（等宽直方图）。



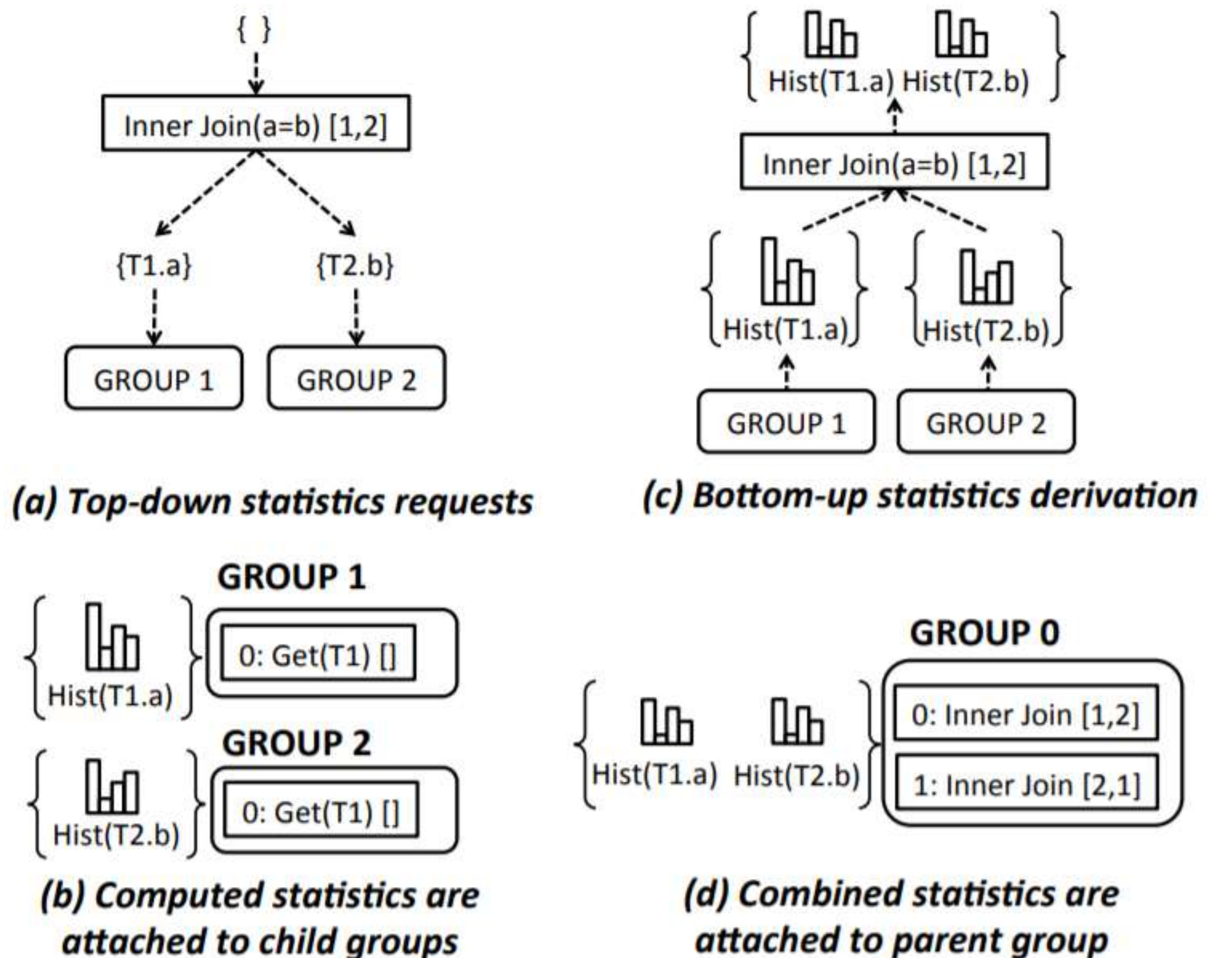
对于单列的范围查询，直方图可以估算选中的行数。只需找出范围条件覆盖的桶，求和桶中的行数。对于不完整覆盖的桶，则按照覆盖的比例**线性推算**；例如 PostgreSQL 例子[44]。

Statistics Derivation

由上文可以发现，单列的柱状图/Cardinality 远不足供代价模型使用。我们还需要

- 通过单列的 Cardinality 推算**多列组合**查询的结果 Cardinality
- 通过单列的 Cardinality 推算 **Join 结果的** Cardinality

例如下图[12]，更基础的统计信息能够逐层向上推导（Derive）出更高层的、包含更多组合的统计信息。



(a) Top-down statistics requests

(c) Bottom-up statistics derivation

(b) Computed statistics are attached to child groups

(d) Combined statistics are attached to parent group

Figure 5: Statistics derivation mechanism

解决上述问题的经典方法基本是假设各列统计独立，即 **AVI: Attribute Value Independence**[21]（以及类似变种）。该方法简单粗暴，但广泛使用，如 PostgreSQL（2015[14]），和许多商业数据库（2001[22]）。

- uniformity: all values, except for the most-frequent ones, are assumed to have the same number of tuples
- independence: predicates on attributes (in the same table or from joined tables) are independent
- principle of inclusion: the domains of the join keys overlap such that the keys from the smaller domain have matches in the larger domain

由此，多列组合查询的 Cardinality 可由各列相乘得到，而 Join 结果的 Cardinality 可用如下[14]方法估算

$$|T_1 \bowtie_{x=y} T_2| = \frac{|T_1||T_2|}{\max(\text{dom}(x), \text{dom}(y))},$$

事实上，How Good Optimizers 论文[14]指出，数据库 Cardinality 估算和实际情况常有几个数量级的差距，总体倾向（严重）低估。

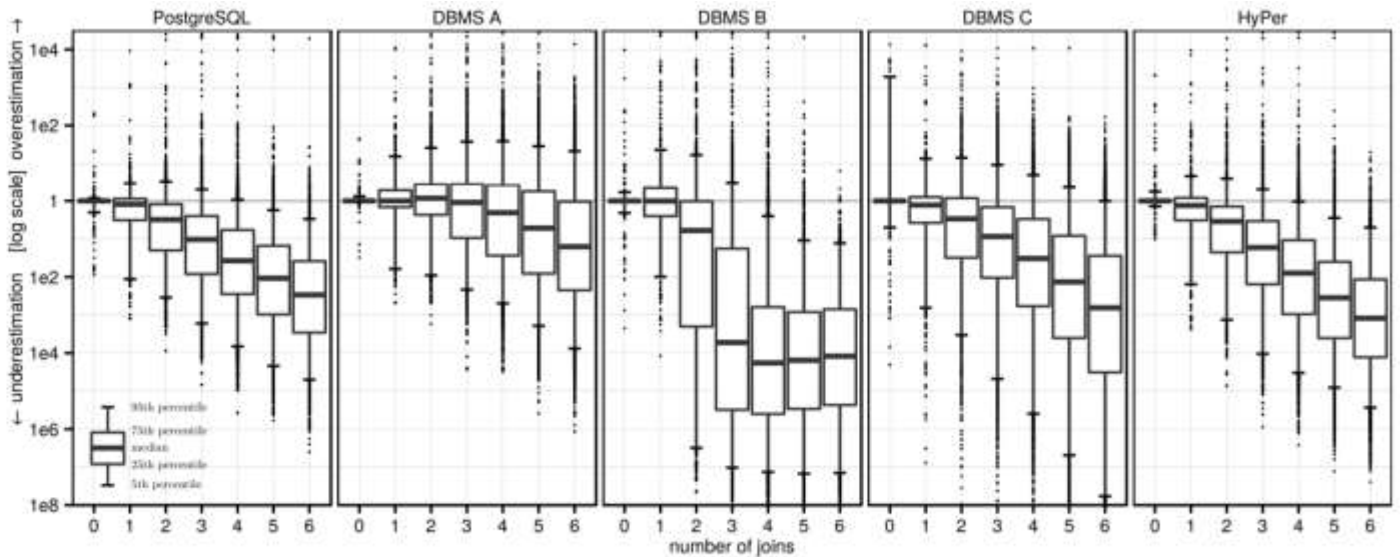


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

为提高 Cardinality 估算的质量，AVI 以外，商业数据库往往支持更复杂更有效方法，如

- Sampling[14] for base table estimation。仅随机抽样 1000 行也能取得不错的估计
- Multi-attribute histograms (Column group statistics[49])。既然单列不够，那么统计多列协同的概率分布
- Query Feedback[16] / Statistics Feedback[48] / LEO Feedback Loop[23]。用查询结果反馈，渐进地改善统计信息

另一个有意思的方向是，在 Database Redbook[4]中提到，Eddies 和 Progressive Optimization。在 **Continuous Stream** 场景下，传统数据库的查询优化和执行步骤不再界限分明。Operator 在执行 Stream 的同时收集统计信息，自适应优化执行计划。与此相应的是 **Dataflow Architecture**。

查询执行（Query Execution）

查询优化器输出物理计划/执行计划，然后交由执行引擎进行查询执行。查询执行已经超出优化器范围，本文简单讲解。

基本名词“Operator”一节中，已介绍查询执行的 Pull、Push 等基本模型。向量化[17]、Compiled Query[20]、利用 SIMD 是内存数据库的热点之一；CPU 成为新的瓶颈。

切分和平衡

并行 Operator 执行常采用水平和垂直的数据切分，在调度（Scheduling）和平衡（Balancing）中常面临如下[3]问题

- **数据倾斜**（Skew）：数据初始分区（Partition）不平衡，或者几经 Operator 处理后的中间结果不再平衡。数据倾斜会使某些任务分区变慢成 Tail[27]，拖慢整体执行
- **处理速度倾斜**：单个处理服务器因硬件差异、临时负载变化、软硬件 Bug 引起的性能下降，变慢成为 Tail，拖慢整体执行

- **Data Locality**：我们希望 Operator 依赖的数据位于同一服务器，避免数据移动开销，即 Locality。对 CPU 核、Cache、NUMA，同样有 Locality 需求。而与 Locality 相矛盾的是，调度器同时希望更细粒度的切分、更广泛的分布，以平衡负载

NUMA 架构
Morsel-Driven Parallelism[18]（HyPer）提供了支持 NUMA 的经典实现。Morsel 与存储系统常见的异步 Task Pipeline 架构相似。此外特点有

- **尊重 NUMA Locality**，一个 CPU 核上的任务尽量存储在同一 NUMA 的内存中，并且优先向同核调度任务
- **Work Stealing** 来平衡负载，空闲 CPU 核"窃取"其它 CPU 核的队列任务。注意 Work Stealing 实际上会打破 NUMA Locality
- HyPer 引入了 **Delay Scheduling**[3]，以防止过于频繁的 Work Stealing。空闲 CPU 核"窃取"前，会等待一小会儿，期待原 CPU 核完成任务。Delay Scheduling[28]也被 YARN[50]采用，据说简单但实际效果意外地好

除 HyPer 的 NUMA-aware 调度方法外，SAP HANA[32P27]使用独立 Watchdog 线程检测工作线程负载，并且动态调整任务分配。而 SQL Server SQLOS[32P32]是"用户态的 OS Layer"，决定任务到线程的分配，线程调度是 Non-preemptive 的（Explicit yield calls）。

分布式 Operator
在基本 Operator 外，分布式数据库引入了新 Operator，以更好地抽象数据移动。它们可由 Property Enforcer 加入。

- **Broadcast[13] / Replicate[25]**：对于需要多张表的操作，如 Join，将一张表复制到另一张表的服务器上。如果另一张表的多个分区位于不同服务器，则需要复制到所有服务器
- **Exchange[3] / Shuffle[13] / Redistribute[12] / Partition[13]**：对于参与 Join 的多张表，按照相同的 Join Key 取 Hash，按照 Hash 值将它们分配到相应多个服务器上。这样相关联的行一定位于相同服务器，可以本地执行 Join 操作
- **Gather[12] / Merge[12]**：对于分散在多个服务器的中间结果，需要将它们收集到单个服务器。收集过程中可同时排序

总结

查询优化器是数据库最复杂、重要、标志性的组件。本文讲述了查询优化器的几大方面；计划搜索、代价模型、统计信息、查询执行，每一门都是独立研究方向。

- 基本名词
- Volcano/Cascades
- 如何搜索计划空间（Plan Enumeration）
- 代价模型（Cost Model）
- 统计信息（Statistics）
- 查询执行（Query Execution）

Volcano/Cascades 是软件工程解决复杂抽象的样板。计划搜索是动态规划的经典应用，特别是如何解决局部最优问题，可借用别处。代价模型则可为各种资源调度系统提供参考。

最终，问题又化归于如何探索广袤[51]的组合空间[33]，寻找最优迭代路径[31]，亦或展开并绘制空间结构[53]。

相关资料

（微信公众号文章不允许贴外部链接，只能将所有引用都加到这里……）

[0] CMU 15-721 Sprint 2020: https://15721.courses.cs.cmu.edu/spring2020/schedule.html

[1] Cascades Optimizer - hellocode: https://zhuanlan.zhihu.com/p/73545345

[2] 揭秘 TiDB 新优化器：Cascades Planner 原理解析: https://pingcap.com/blog-cn/tidb-cascades-planner

[3] OLAP 任务的并发执行与调度 - IO Meter: https://io-meter.com/2020/01/04/olap-distributed

[4] Database Redbook: Chapter 7: Query Optimization: http://www.redbook.io/ch7-queryoptimization.html

[5] How We Built a Cost-Based SQL Optimizer: https://www.cockroachlabs.com/blog/building-cost-based-sql-optimizer

[6] Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe: https://www.eecs.harvard.edu/~kester/files/accesspathselection.pdf

[7] The Volcano Optimizer Generator: Extensibility and Efficient Search: https://15721.courses.cs.cmu.edu/spring2020/papers/19-optimizer1/graefe-icde1993.pdf

[8] The Cascades Framework for Query Optimization: https://15721.courses.cs.cmu.edu/spring2020/papers/19-optimizer1/graefe-ieee1995.pdf

[9] An Overview of Query Optimization in Relational Systems: https://15721.courses.cs.cmu.edu/spring2020/papers/19-optimizer1/chaudhuri-pods1998.pdf

[10] EFFICIENCY IN THE COLUMBIA DATABASE QUERY OPTIMIZER: https://15721.courses.cs.cmu.edu/spring2019/papers/22-optimizer1/xu-columbia-thesis1998.pdf

[11] Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources: https://arxiv.org/pdf/1802.10233.pdf

[12] Orca: A Modular Query Optimizer Architecture for Big Data: https://15721.courses.cs.cmu.edu/spring2016/papers/p337-soliman.pdf

[13] The MemSQL Query Optimizer: A modern optimizer for real-time analytics in a distributed database: http://www.vldb.org/pvldb/vol9/p1401-chen.pdf

[14] How Good Are Query Optimizers, Really?: https://www.vldb.org/pvldb/vol9/p204-leis.pdf

[15] TiDB 源码阅读系列文章（十二）统计信息（上）: https://pingcap.com/blog-cn/tidb-source-code-reading-12

[16] TiDB 源码阅读系列文章（十四）统计信息（下）: https://pingcap.com/blog-cn/tidb-source-code-reading-14

[17] MonetDB/X100: Hyper-Pipelining Query Execution: http://cidrdb.org/cidr2005/papers/P19.pdf

[18] Morsel-Driven Parallelism: A NUMA-Aware Query Evaluation Framework for the Many-Core Age: https://db.in.tum.de/~leis/papers/morsels.pdf

[19] Track Join: Distributed Joins with Minimal Network Traffic: http://www.cs.columbia.edu/~orestis/sigmod14II.pdf

[20] Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask: http://www.vldb.org/pvldb/vol11/p2209-kersten.pdf

[21] Selectivity Estimation Without the Attribute Value Independence Assumption: https://www.vldb.org/conf/1997/P486.PDF

[22] Selectivity Estimation using Probabilistic Models: http://robotics.stanford.edu/~btaskar/pubs/sigmod01.pdf

[23] LEO – DB2's LEarning Optimizer: https://www.vldb.org/conf/2001/P019.pdf

[24] Quasar: Resource-Efficient and QoS-Aware Cluster Management: https://www.csl.cornell.edu/~delimitrou/papers/2014.asplos.quasar.pdf

[25] Query Optimization in Microsoft SQL Server PDW: http://cis.csuohio.edu/~sschung/cis611/MSPDWOptimization_PaperSIG2013.pdf

[26] Synopses for Massive Data: Samples, Histograms, Wavelets, Sketches: https://dsf.berkeley.edu/cs286/papers/synopses-fntdb2012.pdf

[27] The Tail at Scale: https://cacm.acm.org/magazines/2013/2/160173-the-tail-at-scale/fulltext

[28] Delay Scheduling: A Simple Technique for Achieving Locality and Fairness in Cluster Scheduling: http://elmeleegy.com/khaled/papers/delay_scheduling.pdf

[29] Predicting Query Execution Time: Are Optimizer Cost Models Really Unusable: http://pages.cs.wisc.edu/~wentaowu/slides/ICDE-2013.pdf

[30] Forecasting the Cost of Processing Multi-join Queries via Hashing for Main-memory Databases: http://acmsocc.org/2015/posters/socc15posters-final68.pdf

[31] An overview of gradient descent optimization algorithms: https://ruder.io/optimizing-gradient-descent

[32] CMU 15-721: Query Scheduling: https://15721.courses.cs.cmu.edu/spring2020/slides/12-scheduling.pdf

[33] 破晓之钟: 章六十四 囚笼 - 田渊栋: https://zhuanlan.zhihu.com/p/350919763

[34] DBARepublic: Selectivity Vs Cardinality: http://www.dbarepublic.com/2016/02/selectivity-vs-cardinilaty.html

[35] Apache Calcite Tutorial: Calcite-example-CSV: https://calcite.apache.org/docs/tutorial.html

[36] SQL Server Cardinality Estimation: https://docs.microsoft.com/en-us/sql/relational-databases/performance/cardinality-estimation-sql-server

[37] CockroachDB OptGen Rules (Github): https://github.com/cockroachdb/cockroach/tree/master/pkg/sql/opt/norm/rules

[38] TiDB Design Docs IndexMerge.md (GitHub): https://github.com/pingcap/tidb/blob/master/docs/design/2019-04-11-indexmerge.md

[39] TiDB Planner Core Task.go (Github): https://github.com/pingcap/tidb/blob/master/planner/core/task.go

[40] TiDB Cost Variables (Github): https://github.com/pingcap/tidb/blob/master/sessionctx/variable/tidb_vars.go

[41] TiDB Planner Memo (Github): https://github.com/pingcap/tidb/tree/master/planner/memo

[42] PostgreSQL Doc: Cost Variables: https://www.postgresql.org/docs/10/runtime-config-query.html

[43] PostgreSQL Doc: Join Order Enumeration: https://www.postgresql.org/docs/9.5/planner-optimizer.html

[44] PostgreSQL Doc: Row Estimation Examples: https://www.postgresql.org/docs/current/row-estimation-examples.html

[45] Oracle Concepts: Optimization of Joins: https://docs.oracle.com/cd/F49540_01/DOC/server.815/a67781/c20c_joi.htm

[46] Oracle Doc: User-Defined Statistics/Selectivity/Cost: https://docs.oracle.com/cd/B10500_01/appdev.920/a96595/dci08opt.htm

[47] Oracle Doc: Using Zone Maps: https://docs.oracle.com/database/121/DWHSG/zone_maps.htm

[48] Oracle Query Optimizer Concepts: Adaptive Query Plans: Statistics Feedback: https://docs.oracle.com/database/121/TGSQL/tgsql_optcncpt.htm

[49] Oracle Extended Statistics Enhancements: https://oracle-base.com/articles/11g/extended-statistics-enhancements-11gr2

[50] Hadoop: The Definitive Guide, 4th: Chapter 4 YARN: https://www.oreilly.com/library/view/hadoop-the-definitive/9781491901687/ch04.html

[51] 围棋符合现规则终局的所有下法比宇宙中的原子还多 - Zhihu: https://www.zhihu.com/question/342066985/answer/801498759

[52] 揭秘高性能 DolphinDB - Zhihu: https://zhuanlan.zhihu.com/p/40049521

[53] 群论和魔方（P1）- Accela Zhao: https://mp.weixin.qq.com/s/D3ZHMDPgChuCKnMcu95a9A

[54] 分布式系统-分布式事务（P3 完）- Accela Zhao: https://mp.weixin.qq.com/s/FvQO_ZfHbdXKlmRnrt1O7Q