论文分享：How Good Are Query Optimizers, Really?

介绍

Join 重排是查询优化中的一个关键步骤，此阶段优化器通常会通过下面三个步骤枚举可行的 join 顺序，并选择代价最小的 join 顺序：

1. 在计划空间找出一些 join 顺序（join plan enumeration）；
2. 根据统计信息进行估算结果集大小（cardinality estimation）；
3. 把估算结果带入代价模型计算每个 join 顺序的代价（cost model）；

毫无疑问，上述三个优化模块都非常重要，但他们对优化的贡献哪个大，能不能通过实验量化出来，而不只是直觉上和经验主义的衡量？

论文通过创新的实验回答了上述问题，并且产出了一些对优化器设计富有指导性的结论，如：

1. 这三个模块哪个对优化的贡献最大
2. 怎么设计实验才能单独的测试出上述每个模块的贡献
3. 为什么不推荐用 TPC-H，TPC-DS 等数据集来测试优化器能力
4. 和直方图比起来，什么样的统计信息能更好的捕捉数据中的关联性

除此之外你还会看到一些非常有趣的实验结果：

1. 随着 join 表数增加，部分 DB 会出现系统性的低估
2. PostgreSQL 优化器中一个 “负负得正” 的例子
3. 关闭掉 PostgreSQL 的 NestedLoopJoin 后优化效果反而变好

下面会分别介绍论文中的实验方法，对上述三个模块的实验过程和结果分析，最后是总结。
完整的论文请见：How Good Are Query Optimizers, Really?。

实验方法

这节分别从数据集选择，DB 选择，查询构造，实验环境等环节介绍作者怎么设计实验。
阅读这一节后你将知道作者为什么不推荐使用 TPC-H，TPC-DS 等数据集来测试优化器。

数据集选择

许多论文都用 TPC-H，TPC-DS，SSB 等数据集来测试优化器，但作者认为这些数据集不能很好的测出优化器的能力。

原因是这些数据集为了便于扩展，生成时基于过于简单的假设，如均匀独立假设，包含假设等。而真实世界中的数据包含了各种相关性和非均匀分布，这些数据集都不能体现出来。（马上会有一个例子）

最后作者选了 2013 年前的 IMDB 数据，原因如下：

1. 真实而非人为生成，富含关联性；
2. 3.6 GB，最大的两个表分别有 36M 和 15M 行，对优化器测试来说够用；

DB 选择

作者选用 PostgreSQL 进行测试， 原因如下：

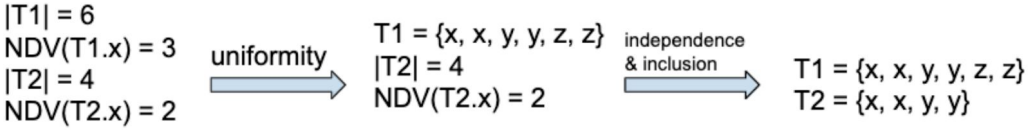
1. 传统架构，包含上述的三个模块
2. 开源免费，易于修改
3. 没有自适应 join，优化器指定的执行顺序不会在执行阶段被再次改变

接下来对 PG join 估算公式进行介绍，同时看一下为什么不推荐使用 TPC-H 等数据集：

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

左边表示 T1 和 T2 在 x 和 y 列进行 join 的结果行数，右边的 |T1| 表示 T1 表的行数，dom(x) 表示这一列的 NDV(number of distinct values)；

该公式基于均匀、独立、包含三个假设，下面是一个例子：



如 T1 和 T2 两个表分别有 6 和 4 行，NDV 分别为 3 和 2，那基于均匀假设，T1 可以写成 {x, x, y, y, z, z} 的模式；再给予独立和包含的假设，T2 可以写成 {x, x, y, y} 的模式；则他们的 join 结果，就是上述公式的计算结果。

可以看出，如果数据基于这三个假设生成，那就刚好命中上述估算规则，使得估算对优化器来说没有任何难度；这就是作者不推荐使用 TPC-H 等数据集测试优化器的原因。

查询构造

基于 IMDB 数据集，作者设计了 33 个查询模板，在每个模板上通过变换一些条件，最后形成了 113 个查询，每个查询包含 3-16 个 join，且会包含如 LIKE 等一些复杂的表达式。

这些查询在逻辑上，也是贴近现实的，如 Q13d，查询所有美国公司出品电影的评分和日期：

```
SELECT cn.name, mi.info, miidx.info
FROM company_name cn, company_type ct,
     info_type it, info_type it2, title t,
     kind_type kt, movie_companies mc,
     movie_info mi, movie_info_idx miidx
WHERE cn.country_code ='[us]'
AND ct.kind = 'production companies'
AND it.info = 'rating'
AND it2.info = 'release dates'
AND kt.kind = 'movie'
AND ... -- (11 join predicates)
```

估算导出和注入

作者把 113 个查询，导入了 5 个 DB（PG、Hyper、3 个商业 DB），然后导出了他们所有的中间估算结果。

如 select * from A, B, C where A.bid=B.id and B.cid=C.id and A.x=5，他的中间结果有：

1. where A.x=5
2. where A.x=5 and A.bid=B.id
3. where B.cid=C.id
4. where A.bid=B.id and B.cid=C.id and A.x=5

然后修改 PG 的代码，支持把这些中间结果注入到 PG 里面，这样可以隔绝掉不同执行器对执行时间的影响，后面会用来进行对比实验。

实验环境

环境上只有一件重要的事情，就是作者把 PG 的内存调整到了 64GB，IMDB 的所有数据都在内存中！

估算测试

这小节作者通过多个实验，测试了估算对查询时间的影响，同时发现了不同 DB 间一些比较有意思的问题。

单表估算测试

作者先进行了单表的基数估算测试，比如：

1. where t.a = 1
2. where t.a > 1 and t.b = 2
3. where t.str like “%abc%”

为了衡量估算效果，引入了 q-error，定义为真实值和估算值的比值，取大于 1 的形式；
如真实值是 100，那估算值为 10 或者 1000，最后的 q-error 都是 10 = 100/10 = 1000/100。

下图是 JOB 中 113 个查询的单表估算，在这 5 个 DB 中的结果：

	median	90th	95th	max
PostgreSQL	1.00	2.08	6.10	207
DBMS A	1.01	1.33	1.98	43.4
DBMS B	1.00	6.03	30.2	104000
DBMS C	1.06	1677	5367	20471
HyPer	1.02	4.47	8.00	2084

Table 1: Q-errors for base table selections

可以发现：

1. 所有 DB 的中位数 q-error 都挺好，接近于最优值 1；
2. 所有 DB 都可能出现误差验证的情况，即 max(q-error) 较大；

作者发现 Hyper 在进行复杂条件估算（如 LIKE 或者多个复杂表达式）时，效果最好。
分析后发现 Hyper 为每个表维护了 1000 行的采样数据，在处理复杂条件时，会直接作用在上面用于估算。
这种维护采样的方式，比起如 PG 的维护单列直方图的方式，能更好捕捉到多列数据关联性。
比如有两列 A 和 B，有隐含的关联条件为 A 恒等于 B*2，现在要求估算 A=2 and B=1：

1. 如果是基于单列直方图，则需要依赖独立性关系假设，估算公式为 Sel(A=2) * Sel(B=1)，这个公式明显忽略了这个隐含的关联信息；
2. 而 Hyper 这种采样的方式，则能很自然的捕捉到其中的关联性；

Join 多表估算测试

接下来测试各个 DB 对多表 join 的估算：

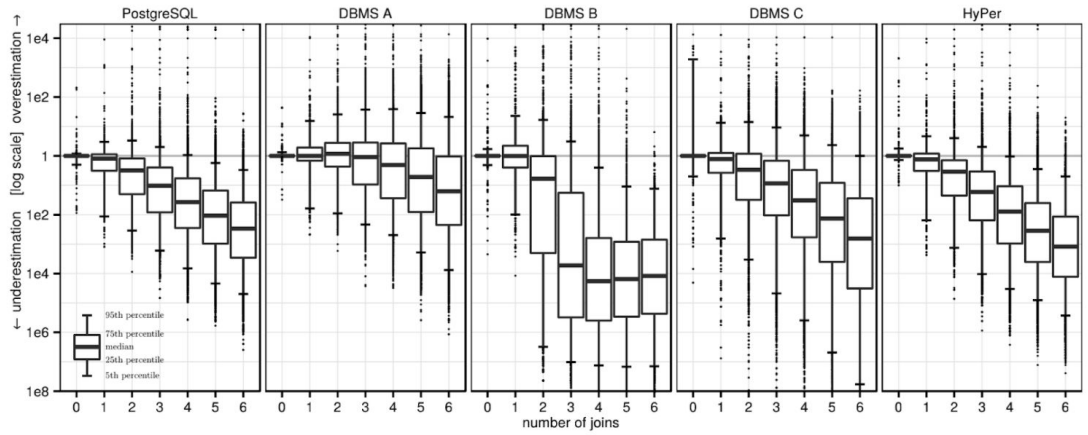


Figure 3: Quality of cardinality estimates for multi-join queries in comparison with the true cardinalities. Each boxplot summarizes the error distribution of all subexpressions with a particular size (over all queries in the workload)

横坐标为 join 的个数，纵坐标为估算结果和真实值的比（取 log）；

可以得到一个很有趣的结论：所有的 DB，随着 join 个数增加，都会系统性的低估。

推测其内在原因为：随着 join 个数增加，数据之间的关联性增强，而如 PG 这样基于独立相关性等假设的 join 估算公式，不能发现其内在关联性。

其中 DBMS A 的结果稍好，推测为内置了一个衰减因子，随着 join 数量增加，抵消一部分独立性关系假设的影响。

TPC-H Join 估算测试

接下来是 TPC-H 的 join 估算测试：

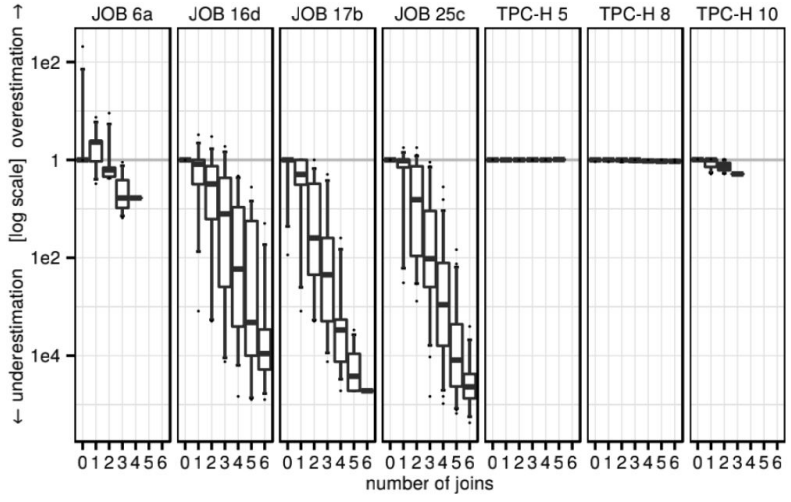


Figure 4: PostgreSQL cardinality estimates for 4 JOB queries and 3 TPC-H queries

可以发现，TPC-H 对 PG 的优化器构成不了什么挑战，再一次验证了之前说的结论。

NDV 注入测试

根据一开始对 PG join 估算的介绍，我们知道其中最重要的一个参数就是 NDV，为了验证 NDV 对 PG 估算的具体影响，我们将 JOB 查询的真实 NDV 注入到了 PG 当中，然后进行了测试：

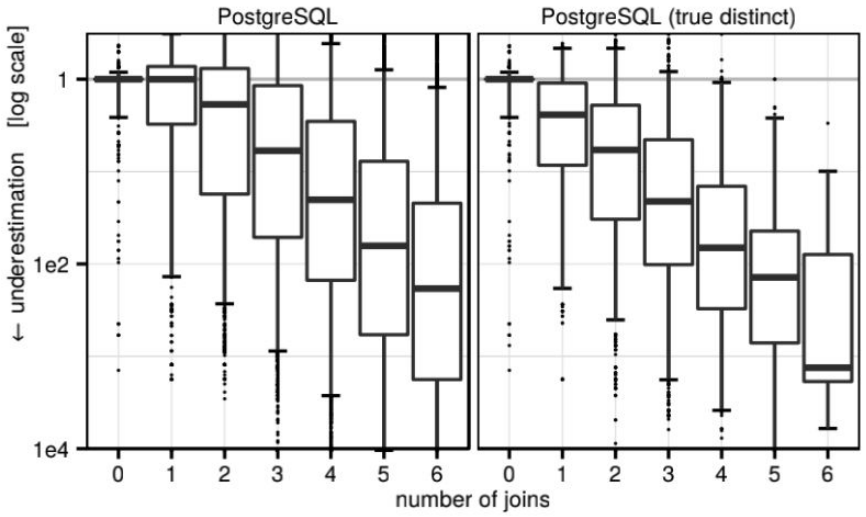


Figure 5: PostgreSQL cardinality estimates based on the default distinct count estimates, and the true distinct counts

很有趣，发现注入真实 NDV 后，估算结果反而变差了！

这是因为：

1. 如前文所说，随着 join 数量增加，PG 会系统性低估；
2. PG 的统计信息采集方式会造成 NDV 被低估；
3. 根据 PG 的 join 估算公式，低估的 NDV 造成高估的 join 估算结果；

上面 3 个原因，使得原本被低估的 NDV，反而能够修正一些系统性低估的误差，出现了“负负得正”的效果！

基数估算误差对查询时间影响测试

在这个实验中，将不同 DB 的估算结果（包括真实基数），都注入到 PG 中，然后运行并记录时间；这样可以屏蔽掉不同 DB 执行器的影响，只验证他们估算误差对执行时间的影响；然后将运行时间和最优计划的运行时间（通过注入真实基数得到）做比，得到下面结果：

	<0.9	[0.9,1.1)	[1.1,2)	[2,10)	[10,100)	>100
PostgreSQL	1.8%	38%	25%	25%	5.3%	5.3%
DBMS A	2.7%	54%	21%	14%	0.9%	7.1%
DBMS B	0.9%	35%	18%	15%	7.1%	25%
DBMS C	1.8%	38%	35%	13%	7.1%	5.3%
HyPer	2.7%	37%	27%	19%	8.0%	6.2%

然后有下面的结论：

1. 和最优计划（使用真实基数）比起来，所有 DB 中大部分查询执行时间都显著变长，也就是比值 >= 1.1 的部分，这反向展示出估算误差对查询时间的影响；
2. 少数查询的耗时降低了，也就是 <0.9 的部分，这是由于 cost model 误差引起；
3. 所有 DB 都有部分查询耗时严重增加，也就是 >= 10 的部分；

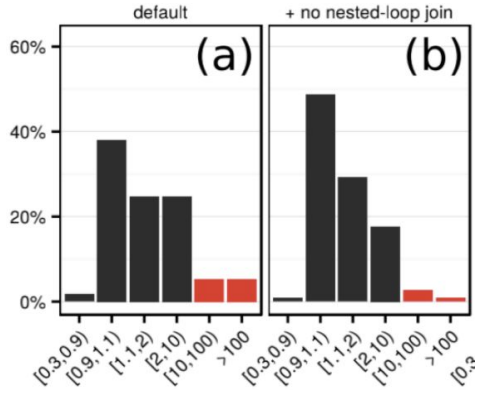
过于依赖估算的风险

接着分析上述实验中，耗时增加严重的那些查询，也就是 >=10 的部分，发现大部分都是由于把 HashJoin 选错成 NestedLoopJoin 造成的。

分析 PG 代码，发现它的 join 选择只考虑 cost，相当于完全相信估算的准确性，而不考虑风险；HJ 的复杂度是 O(N)，NLJ 的复杂度是 O(N^2)，如果 HJ 的代价是 100001，NLJ 代价是 100000， 也会选择 NLJ。

结合之前之前所说，随着 join 数量增加，PG 容易出现系统性低估，这就使得 PG 容易出现误选 NLJ 的情况，由于其复杂度高，使得执行时间大大增加。

作者认为这样的策略是低收益高风险的，应该避免，如下图，关闭掉 NLJ 后，耗时严重的查询变少了：



还有一部分查询较差，是由于估算偏低，导致 HashJoin 事先分配的 hash table 过小，导致执行时冲突大，延长了执行时间。

在 PG 之后的版本中，执行器实现了自适应的 rehashing 功能，作者把这个功能移植到了目前实验的 PG 中，又进行了一次实验：

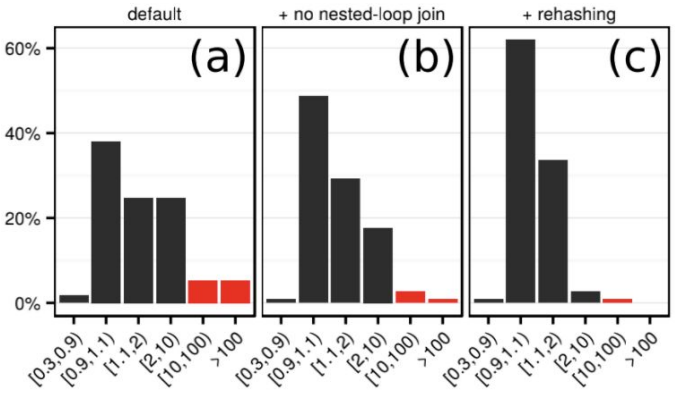


Figure 6: Slowdown of queries using PostgreSQL estimates w.r.t. using true cardinalities (primary key indexes only)

可见效果又变好了，执行阶段的自适应操作，能够缓解优化器错误判断带来的影响。

代价模型

本节作者对比了 3 种不同的代价模型，尝试衡量出代价模型对优化效果的影响。

实验用的三个代价模型

在这个小节，作者使用了 3 个代价模型，来测试他对执行耗时的影响。

第一个代价模型是 PG 原生的，其特点为：

1. 面向磁盘，主要考虑 CPU 和磁盘 I/O
2. 通过不同的权重把 CPU 和磁盘 I/O 结合起来作为代价

因为测试中所有数据都在内存中，于是第二个就是在 PG 模型为内存调整：

1. PG 原生代价参数认为磁盘操作比 CPU 操作贵 400 倍
2. 作者把这个代价参数除以了 50，相当于任务内存操作比 CPU 操作贵 8 倍

第三个模型(Cost-Main-Mem)特别简单，只考虑了算子处理的数据行数，大致如下：

$$C_{\mathrm{mm}}(T) = \begin{cases} \tau \cdot |R| & \text{if } T = R \vee T = \sigma(R) \\ |T| + C_{\mathrm{mm}}(T_1) + C_{\mathrm{mm}}(T_2) & \text{if } T = T_1 \bowtie^{\mathrm{HJ}} T_2 \\ C_{\mathrm{mm}}(T_1) + & \text{if } T = T_1 \bowtie^{\mathrm{INL}} T_2, \\ \quad \lambda \cdot |T_1| \cdot \max(\frac{|T_1 \bowtie R|}{|T_1|}, 1) & (T_2 = R \vee T_2 = \sigma(R)) \end{cases}$$

比如 HashJoin 的代价就是左右儿子的代价，加上产出的行数，也就是 |T|；

Selection 和 IndexJoin 的两个参数，主要是用来调整表扫描，和索引扫描的代价权重；

代价模型试验

作者在使用估算，和使用真实基数，两种情况下，对三种模型进行了试验，并对试验结果使用了线性回归，如下图：

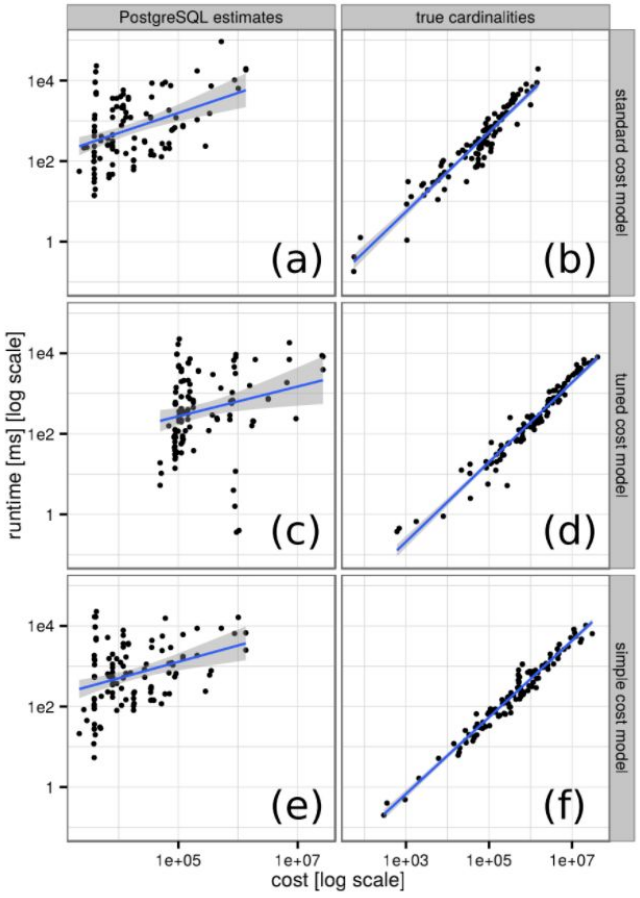


Figure 8: Predicted cost vs. runtime for different cost models

上图中，横坐标是代价，纵坐标是执行时间，蓝色的线是线性回归的结果，灰色部分是抖动。

首先需要知道，什么样的代价函数，是好的代价函数？显而易见它要满足下面两个性质：

1. 递增：也就是更高的代价，对应更长的执行实际；
2. 误差小：也就是观测点，尽量分布在代价函数的两侧，抖动不要太大；

作者对试验结果使用了线性回归，而不是更复杂的函数，原因是在使用真实基数的情况下，线性函数能很完美的满足上面两个性质了（拟合度足够高），我们可以把这个线性回归的结果，当做我们的代价模型函数。

分析上图，我们可以发现：

1. 在没有使用真实基数的情况下，调整代价模型对执行时间的影响不大，a，c，e 中的查询执行时间并没有显著的降低；对比 Figure 6，可见和估算比起来，代价模型的影响相对小很多；
2. 使用真实基数后，d 和 f 中查询时间的几何均数，分别比 b 中快 41% 和 34%，可见代价模型的调整还是有一定收益；

Join 计划空间

这小节作者通过多个实验，探究了 join 顺序，bushy-tree 结构及不同的 join 算法对优化效果的影响。

Join 顺序的影响

作者使用了 Quickpick 算法，为每个查询，生成了 10000 种 Join 顺序，看他们的执行时间。

1. Quickpick 相当于每次随机选两个 join 节点连接，用来生成随机的 join 顺序；
2. 有了 join 顺序，作者并没有真实的执行，因为从上小节的实验来看，在使用真实基数的情况下，代价模型已经能比较准确的反映执行时间，于是作者直接计算了他们的代价，作为标准来衡量执行时间；

下面是实验结果：

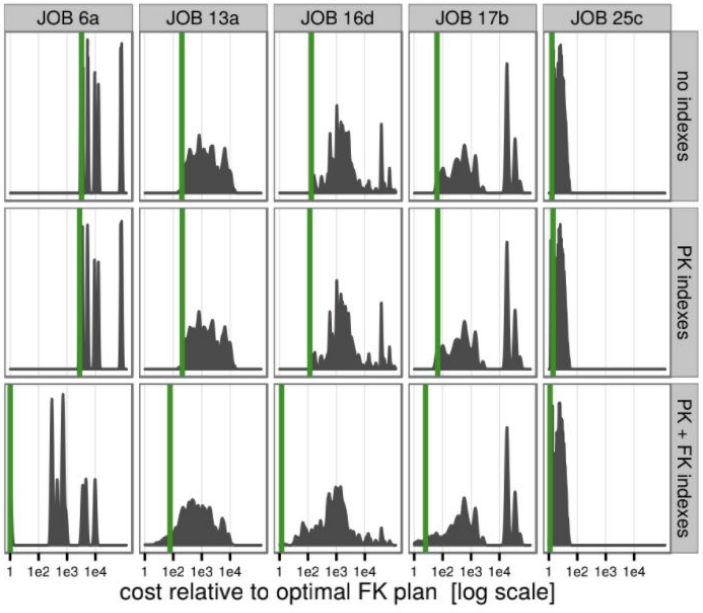


Figure 9: Cost distributions for 5 queries and different index configurations. The vertical green lines represent the cost of the optimal plan

横坐标为代价（相当于执行时间），纵坐标表示分布，可见：

- 观察所有图的分布，最好的 join 顺序比最差的好处几个数量级，可见 join 顺序对执行时间的影响巨大；
- 对比第三排和前两排，可知更多的 index 能有效降低多表查询的时间；

Bushy Tree 测试

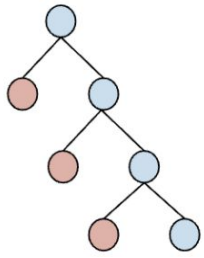
这一小节实验主要想看 bushy-tree 结构对结果的影响。
PG 的 join 重排算法基于 DP，可以遍历到所有的 join 顺序，作者修改它，只遍历 left-deep, right-deep, zig-zag 三种模式，即不包含 bushy-tree。
用这三种模式的遍历结果，和最优的 join 顺序（通过原有的 DP 算法得到，可能是 bushy-tree）做比，得到下面的结果：

	PK indexes			PK + FK indexes		
	median	95%	max	median	95%	max
zig-zag	1.00	1.06	1.33	1.00	1.60	2.54
left-deep	1.00	1.14	1.63	1.06	2.49	4.50
right-deep	1.87	4.97	6.80	47.2	30931	738349

Table 2: Slowdown for restricted tree shapes in comparison to the optimal plan (true cardinalities)

在只有主键索引的情况下，不包含 bushy-tree 结构的 zig-zag 模式的空间遍历，已经能取得较好的结果了，对最差的 95% 分位的查询而言，仅比最优解慢 6%。

right-deep 比 left-deep 差的原因和 PG 的 hash join 实现有关，如下：



对于一个 N 表 join 的查询而言，right-deep 的模式需要构造 N-1 个 hash 表来存中间结果，上图的红色部分。

Join 重排算法测试

为了对比不同 join 重排算法对结果的影响，作者选用了 3 个算法进行对比测试：

1. PG 原本的 DP 算法：遍历全部的空间，能选取到最优解
2. Quick-1000：对每个查询随机产生 1000 种 join 顺序，然后选取其中代价最小的
3. 贪心算法：每次选估算结果最少的两个表进行 join，直到所有表被 join 起来

同样的，作者记录不同算法获取的计划，和最优计划代价的比值，实验结果如下：

	PK indexes						PK + FK indexes					
	PostgreSQL estimates			true cardinalities			PostgreSQL estimates			true cardinalities		
	median	95%	max	median	95%	max	median	95%	max	median	95%	max
Dynamic Programming	1.03	1.85	4.79	1.00	1.00	1.00	1.66	169	186367	1.00	1.00	1.00
Quickpick-1000	1.05	2.19	7.29	1.00	1.07	1.14	2.52	365	186367	1.02	4.72	32.3
Greedy Operator Ordering	1.19	2.29	2.36	1.19	1.64	1.97	2.35	169	186367	1.20	5.77	21.0

Table 3: Comparison of exhaustive dynamic programming with the Quickpick-1000 (best of 1000 random plans) and the Greedy Operator Ordering heuristics. All costs are normalized by the optimal plan of that index configuration

下面进行分析：

1. 当使用真实估算时，DP 能 100% 得到最优计划，因为他能完全遍历所有空间；
2. 使用真实估算后，各个算法的准确率都大幅度提升，可见真实估算带来的收益，比调整 join 算法的收益大；

总结

论文通过创新的实验，量化了优化器各模块的收益，并产出了一些富有指导性的结论：

1. 优化收益：基数估算 > 计划空间 > 代价模型
2. TPC-H 的数据分布过于简单，对优化器的基数估算形成不了挑战，推荐使用 JOB
3. 像 Hyper 那种为 table 维护采样的方式，能简单且鲁棒的探测多列数据中的关联性
4. 优化器最好考虑估算误差带来的风险，避免低收益高风险的优化
5. 如果可能，尽量在执行期间引入自适应技术，来缓解掉优化器估算错误的影响

完整的论文请见：How Good Are Query Optimizers, Really?。