

Обработка и исполнение запросов в СУБД (Лекция 7)

Колоночные СУБД: схемы для OLAP; выполнение соединений в колоночных СУБД; сравнение с классическими СУБД

v6

Георгий Чернышев

Высшая Школа Экономики

chernishev@gmail.com

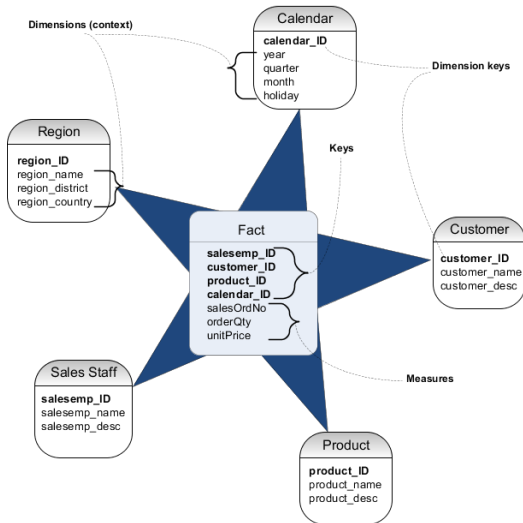
14 октября 2020 г.

Про OLAP “на пальцах”

- OLAP базы данных имеют специальную схему: Star Schema, Snowflake Schema, ...
- Таблица фактов: хранит основные записи
 - Большая, на несколько порядков больше таблиц измерений;
 - Широкая, сотни атрибутов;
 - Много FK, кроме них есть еще меры;
- Таблицы измерений:
 - Количество записей мало, не широкие;
 - Данные изменяются редко, в основном добавления;

На самом деле всё сложнее, м.б. потом отдельно поразбираем OLAP.

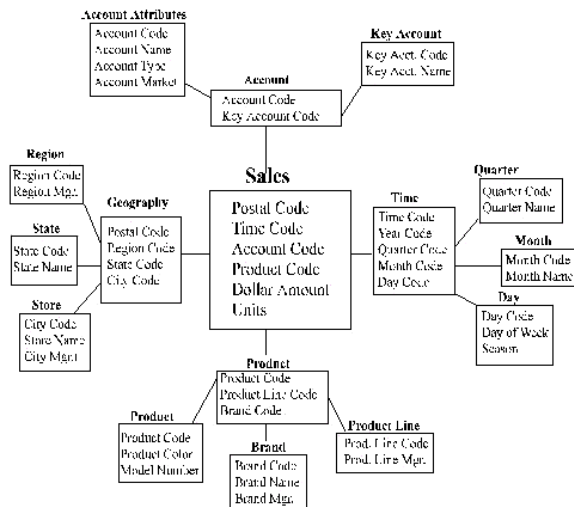
Схема “звезда”, пример



1

¹Изображение взято из https://docs.infor.com/help_lawson_cloudsuite_10.0/topic/com.lawson.help.reporting/com.lawson.help.bpwapg-w_10.4.0/L55461185818015.html

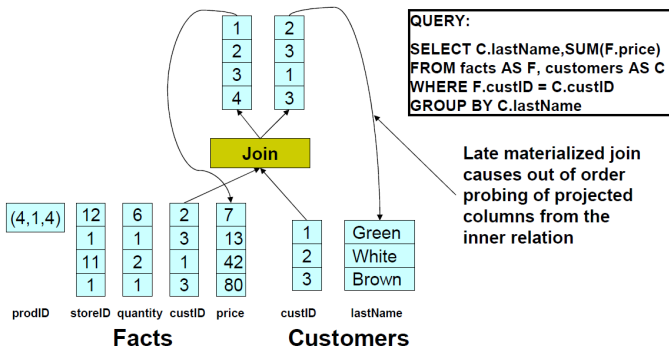
Схема “снежинка”, пример



2

²Изображение взято из <http://www.informix.com.ua/articles/rolap/rolap.htm>

Итоги прошлой лекции



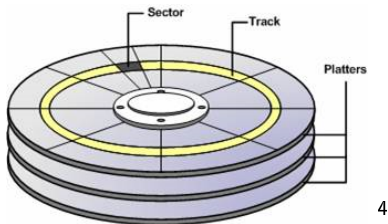
3

Соединения и поздняя материализация “не дружат”:

- По одной из колонок теряется отсортированность;
 - Вычитка несортированной колонки очень дорога;
- нужны другие операторы соединения, специализированные для этих нагрузок

³Изображение взято из [Harizopoulos et al., 2009]

Жесткий диск



Затраты времени:

- Seek time + Rotational latency = около 2.5 ms
- Command processing time = мало
- Settle time = мало

Считать с диска 100 элементов случайным доступом — 250ms.

Последовательно: 2.5ms + 0.1ms.

→ надо избегать случайного доступа

⁴ Изображение взято из <http://www.appleplexsoft.com/glossary/hard-disk.html>

→ нужны эффективные операторы соединения

Как? Использовать особенности OLAP, примеры:

- TPC-H [TPC-H Specification, 2011];
- Star Schema Benchmark
[Star Schema Benchmark Specification, 2009] — модифицированный TPC-H, исправляет недостатки TPC-H;
- Другие: TPC-DS.

Star Schema Benchmark

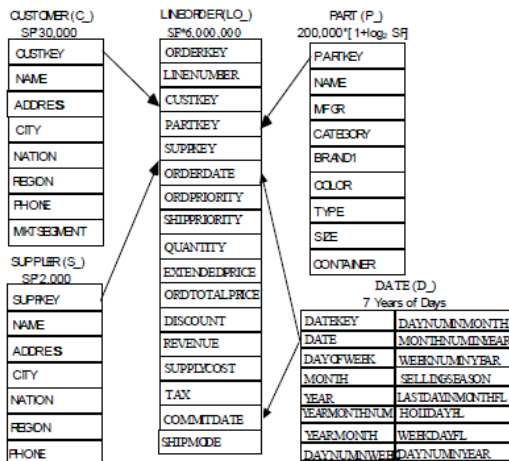


Figure 1.2 SSB Schema

5

Невидимое соединение

```
1 SELECT
2     c.nation, s.nation, d.year, sum(lo.revenue) as revenue
3 FROM customer AS c, lineorder AS lo,
4     supplier AS s, dwdate AS d
5 WHERE lo.custkey = c.custkey
6     AND lo.suppkey = s.suppkey
7     AND lo.orderdate = d.datekey
8     AND c.region = 'ASIA'
9     AND s.region = 'ASIA'
10    AND d.year >= 1992 and d.year <= 1997
11 GROUP BY c.nation, s.nation, d.year
12 ORDER BY d.year asc, revenue desc;
```

Как вычислять запрос?

Стратегии:

- Классический подход (ранняя материализация):
 - Предикаты спустить вниз на уровень измерений;
 - Упорядочить соединения по убыванию селективности;
 - Результат сгруппировать и посчитать агрегатные функции;
- Поздняя материализация:
 - Применяем `c.region = 'ASIA'`, получаем ключи у Customer;
 - Соединяем с Fact Table, получаем позиции для LineOrder и Customer;
 - Кто-то из них будет не отсортирован (обычно dimension) : (
 - Забираем из customer атрибут `c.nation` по позициям (не отсортированы) и отсортированные данные из LineOrder (`supplier key, order date, revenue`);
 - Аналогично соединяем `supplier` и `date`;

Как вычислять запрос? II

Оба подхода плохи:

- Классический подход (ранняя материализация). Нет выигрышей от поздней материализации:
 - нельзя поэкономить на чтении с диска;
 - нет улучшения локальности кешей;
 - нет возможности оперировать над сжатыми данными.
- Поздняя материализация:
 - Значения в `dimensions` не отсортированы, придется вытаскивать их не по порядку. Что весьма дорого.

Выход — оператор Invisible Join: поздняя материализация и минимизация количества значений читаемых не по порядку.

Invisible Join I

Идея: переписать соединения в предикаты на FK колонки в таблице фактов. Шаги алгоритма:

- 1 Применяем предикаты к таблицам измерений, получаем ключи;
- 2 На ключах строится хеш-таблица (т.е. будем проверять правда ли что строка с данным значением ключа удовлетворяет предикату);

Apply region = 'Asia' on Customer table

custkey	region	nation	...
1	Asia	China	...
2	Europe	France	...
3	Asia	India	...

Hash table
with keys
1 and 3

Apply region = 'Asia' on Supplier table

supkey	region	nation	...
1	Asia	Russia	...
2	Europe	Spain	...

Hash table
with key 1

Apply year in [1992,1997] on Date table

dateid	year	...
01011997	1997	...
01021997	1997	...
01031997	1997	...

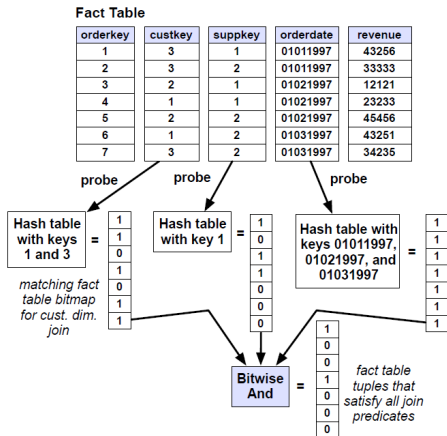
Hash table with
keys 01011997,
01021997, and
01031997

6

⁶ Изображение взято из [Abadi et al., 2008]

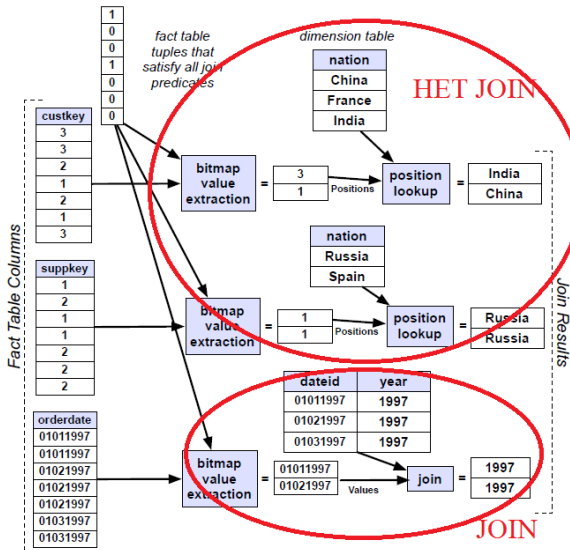
Invisible Join II

- Идем по таблице фактов, опрашивая хеш-таблицы, от каждой получаем список позиций (записи удовлетворяющие предикату);
- Полученные списки позиций пересекаем;



7

Invisible Join III (схема)



8

Invisible Join III (описание)

- ⑤ Из таблицы фактов получаем позиции в измерениях;
- ⑥ Идем в таблицы измерений и забираем значения;
 - ① Если ключи в измерении отсортированы и начинаются с 1 то можно делать position lookup, фактически как массиве;
 - ② Колонка в таблицах измерений обычно маленькая, помещается в L2 кеш, поэтому всё быстро;
 - ③ Если ключи не отсортированы и не плотны, то придется соединять (последний случай на схеме в предыдущем слайде);
 - ④ Соединять не так плохо как кажется: будет только один результат в каждой таблице измерений, значит будет одинаковое количество результатов в каждой таблице измерений, значит соединения можно параллелить и комбинировать результат после!
 - ⑤ Высокая селективность помогает выбрать меньше на этом шаге;

Оставляем таблицу фактов отсортированной всегда!

Алгоритм соединения с использованием Join Index. Делаем интервальное фрагментирование, затем каждый фрагмент обрабатываем.

Делает один последовательный проход по каждому из отношений и два последовательных просмотра временного файла. При этом, размер вспомогательного файла составляет половину от размера Join Index.

Алгоритм требует наличия свободной памяти (блоков) порядка квадратного корня из размера (в блоках) из наименьшего отношения.

Join Index: вспомогательная структура данных, позволяющая ускорить соединение. Устройство: список пар (id записи в первой таблице, id записи во второй).

Старый метод [Li and Ross, 1999]. Имеем R_1 , R_2 , J — join index (отсортирован по R_1 tuple id)

Три фазы:

- J1: выбираем $y - 1$ значений (tuple ids) из R_2 , будем фрагментировать R_1 , пытаемся максимально равно. Каждый фрагмент будет иметь output file buffer и temporary file buffer.
- J2: последовательно идем по J и R_1 как в merge join; если есть совпадение:
 - Атрибуты R_1 , что нужны для результата пишутся в output file buffer фрагмента;
 - R_2 tuple id пишутся в temporary file buffer фрагмента;

При переполнении фрагмента сбрасываем его на диск.

- Закончив, сбрасываем все на диск, получили набор JR_1 и temporary file;

Jive Join пример 1

Student	Course	Course	Instructor
Smith ¹	101	101	Green
Smith ²	109	102	Yellow
Jones	104	103	Green
Davis ¹	102	104	White
Davis ²	105	105	Evans
Davis ³	106	106	Alberts
Brown	102	106	Beige
Black	103	108	Red
Frick	107	109	Grey

Relation *Student* Relation *Course*

Student	Course	Instructor	Student tuple id	Course tuple id
Smith ¹	101	Green	1	1
Smith ²	109	Grey	2	9
Jones	104	White	3	4
Davis ¹	102	Yellow	4	2
Davis ²	105	Evans	5	5
Davis ³	106	Alberts	6	6
Davis ³	106	Beige	6	7
Brown	102	Yellow	7	2
Black	103	Green	8	3

Join Result Join Index

9

⁹ Изображение взято из [Li and Ross, 1999]

Join Join пример II

Шаги J1 и J2:

Фрагменты: а) $id < 3$, b) $3 \leq id < 6$ и с) $6 \leq id$.

(a) $id < 3$		(b) $3 \leq id < 6$		(c) $6 \leq id$	
Smith ¹	1	Jones	4	Smith ²	9
Davis ¹	2	Davis ²	5	Davis ³	6
Brown	2	Black	3	Davis ³	7
$JR_1(a)$	$\overline{Temp(a)}$	$JR_1(b)$	$\overline{Temp(b)}$	$JR_1(c)$	$\overline{Temp(c)}$

10

Слева JR_1 , справа temporary output file.

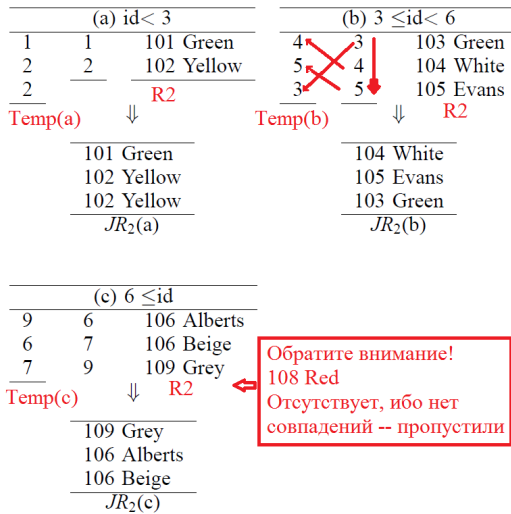
¹⁰ Изображение взято из [Li and Ross, 1999]

J3 (начало): читаем по отдельности набор temporary file

- грузим весь фрагмент в память, сортируем в памяти R_2 tuple id, по возрастанию, удаляем дубликаты + храним исходную версию во временном файле;
- читаем по порядку записи из R_2 , если совпало – пишем в JR_2 ;
- место определяем по старому порядку, строим над ним хеш, как вариант;

Join Join пример II

Шаг J3 (начало):



11

¹¹ Изображение взято из [Li and Ross, 1999]

Join пример III

Шаг J3 (продолжение): склеиваем $JR_1(x)$ и $JR_2(x)$, последовательно берем каждую пару фрагментов и выполняем “склейку”.

	Student	Course	Instructor	
	Smith ¹	101	Green	
$JR_1(a)$	Davis ¹	102	Yellow	$JR_2(a)$
	Brown	102	Yellow	
	Jones	104	White	
$JR_1(b)$	Davis ²	105	Evans	$JR_2(b)$
	Black	103	Green	
	Smith ²	109	Grey	
$JR_1(c)$	Davis ³	106	Alberts	$JR_2(c)$
	Davis ³	106	Beige	

12

¹² Изображение взято из [Li and Ross, 1999]

Jive join: характеристика и итоги

Плюсы:

- Почти весь ввод/вывод последовательный;
- Блок читается с диска только если есть запись участвующая в соединении;
- J читается единожды;
- Временных файлов размером в половину от J сначала пишут, потом читают;
- Однопроходен по R_1 и R_2 (если хватит памяти);
- Со skew можно побороться (см. equi-depth, Лекция 4);

Минусы:

- Ломается pipelining, ждем конца J_1 и J_2 ;
- Нужен диск на промежуточные результаты — $JR_1 + t.$ output files;
- Есть ограничения по оперативной памяти на R_2 (меньшее отношение), расчеты в статье;

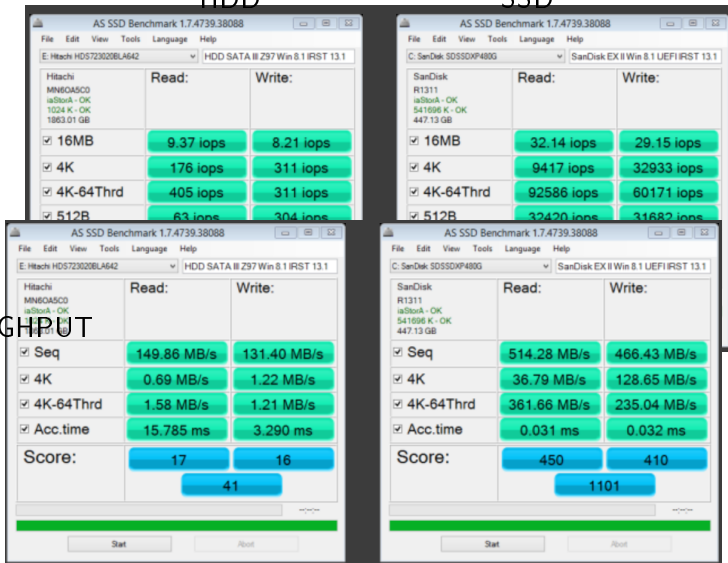
Про диск II (latency, iops, throughput)

HDD

SSD

IOPS

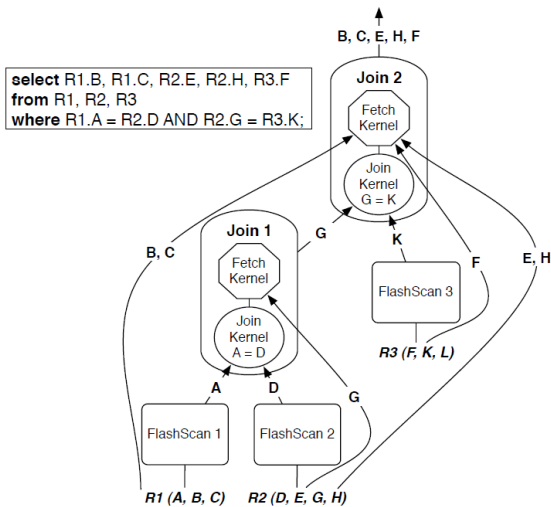
THROUGHPUT



13

13 Изображения взяты из <http://www.thessdreview.com/featured/ssd-throughput-latency-iops-explained/>

FlashJoin “на пальцах” I



14

FlashJoin “на пальцах” II

Идея: используем быстрые random reads на SSD.

- Multi-way equi-join, реализованный как последовательность бинарных;
- Каждый бинарный состоит из join kernel и fetch kernel;
- Join kernel выдает join index, содержит $(RID, attr_1, attr_2)$;
- Fetch kernel по RID считывает нужный атрибут для следующего соединения;
- Поздняя материализация результата;
- Меньше читаем с диска, меньше ходит между операторами (J) – более эффективен по памяти, меньше вероятность многопроходности;

Как использовать?

Проблема: первый случай в Invisible Join III, position lookup идет не последовательно (и пусть отношение больше памяти), производительность страдает от дисковой seek latency.

Идея:

- Добавить колонку, которая занумерует (1, 2, 3, ...) колонку вычитки из таблицы измерения;
- Отсортировать по второй колонке (позициям на поиск);
- Сделать соединение, используя последовательное чтение;
- Отсортировать по первой колонке и “подклеить”.

Подробный алгоритм смотрите в [Harizopoulos et al., 2009].

Поздняя материализация и соединения, итог

- если применять “В лоб” — всё портится, медленнее в 2 раза по сравнению с ранней из-за “дергания” диска;
- Используя трюки наподобие Invisible Join, Jive Join, Flash Join, Radix Cluster/Deccluster — для некоторых запросов можно добиться что поздняя материализация станет в 2 раза лучше ранней.

Это исследования, индустрия же не верит (-ла в 2009, как минимум): используют LM планы только когда неотсортированная колонка вкладывается в память целиком (можем дешево отсортировать).

Около 2013 индустрия полностью разочаровалась в LM, кажется что Vertica (наследник C-Store) убрала ее и заменила на SIP (sideways information passing). Однако, диски стали в этот период сильно лучше и поэтому на мой взгляд будущее у этой идеи есть.

Сравнение row-store и column-store¹⁵

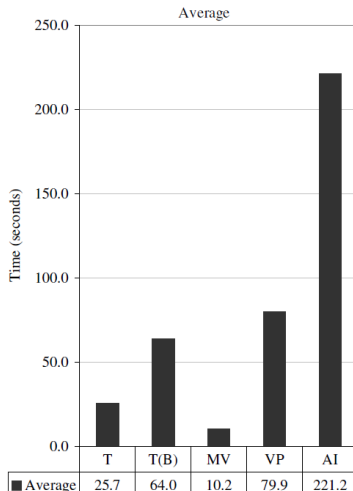
Эмуляция колоночной СУБД в System-X [Abadi et al., 2008]:

- VP: Полное вертикальное фрагментирование (ключ, атрибут);
 - Соединять с помощью NJ, пытались и кластеризованный индекс — хуже.
- AI: Index-only планы (каждый атрибут проиндексирован), так чтобы не читать таблицы вообще;
 - Дополнительный некластеризованный индекс.
- MV: Материализованное представление для каждого запроса, только нужные атрибуты, без prejoined tables.

Против:

- T: Дефолтные планы, которые могут использовать битмап индексы и блум фильтры, если сочтет нужным;
- T(B): Дефолтные планы с битмап индексами.

¹⁵есть на русском citforum.ru/database/articles/column_vs_row_store/



16

Взяли query 2.1 из Star Schema Benchmark (в C-Store выполнялся 5.7с).

¹⁶ Изображение взято из [Harizopoulos et al., 2009]

Какие особенности CS дают большее преимущество?

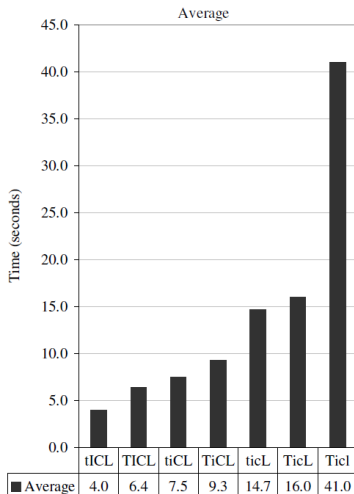
Особенности [Abadi et al., 2008]:

- Одна запись vs несколько записей;
- Invisible Join;
- Сжатие;
- Поздняя материализация;

Легенда (конфигурация C-Store):

- T=tuple-at-a-time processing, t=block processing;
- I=invisible join enabled, i=disabled;
- C=compression enabled, c=disabled;
- L=late materialization enabled, l=disabled.

Какие особенности CS дают большее преимущество: сравнение



17

Большинство изображений взяты из оригиналов статей или прямо из слайдов [Harizopoulos et al., 2009].



Dimitris Tsirogiannis, Stavros Harizopoulos, Mehul A. Shah, Janet L. Wiener, and Goetz Graefe. 2009. Query processing techniques for solid state drives. In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (SIGMOD '09), Carsten Binnig and Benoit Dageville (Eds.). ACM, New York, NY, USA, 59-72. DOI=<http://dx.doi.org/10.1145/1559845.1559854>



Zhe Li and Kenneth A. Ross. 1999. Fast joins using join indices. The VLDB Journal 8, 1 (April 1999), 1–24. DOI=<http://dx.doi.org/10.1007/s007780050071>



Star Schema Benchmark. Revision 3, June 5, 2009 Pat O'Neil, Betty O'Neil, Xuedong Chen



Daniel J. Abadi, Samuel R. Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In Proceedings of the 2008 ACM SIGMOD international conference on Management of data (SIGMOD '08). ACM, New York, NY, USA, 967-980. DOI=<http://dx.doi.org/10.1145/1376616.1376712>



TPC BENCHMARK(TM) H (Decision Support) Standard Specification Revision 2.14.2



Daniel Abadi, Peter Boncz, Stavros Harizopoulos. The Design and Implementation of Modern Column-Oriented Database Systems. Foundations and Trends(R) in Databases Vol. 5, No. 3 (2012) 197–280



Stavros Harizopoulos, Daniel Abadi, Peter Boncz. Column-Oriented Database Systems. VLDB 2009 Tutorial (slides).