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

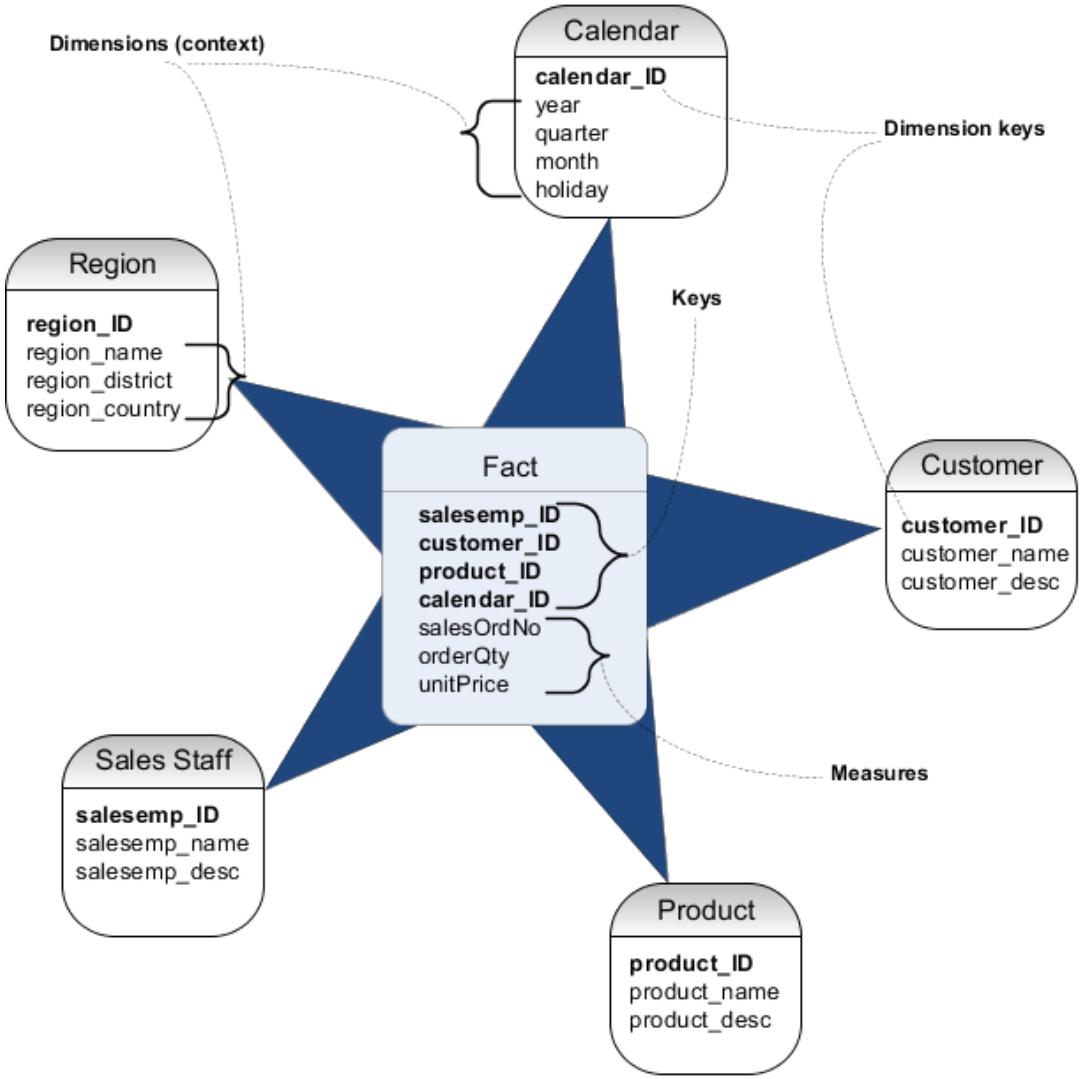
14 октября 2020 г.

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.

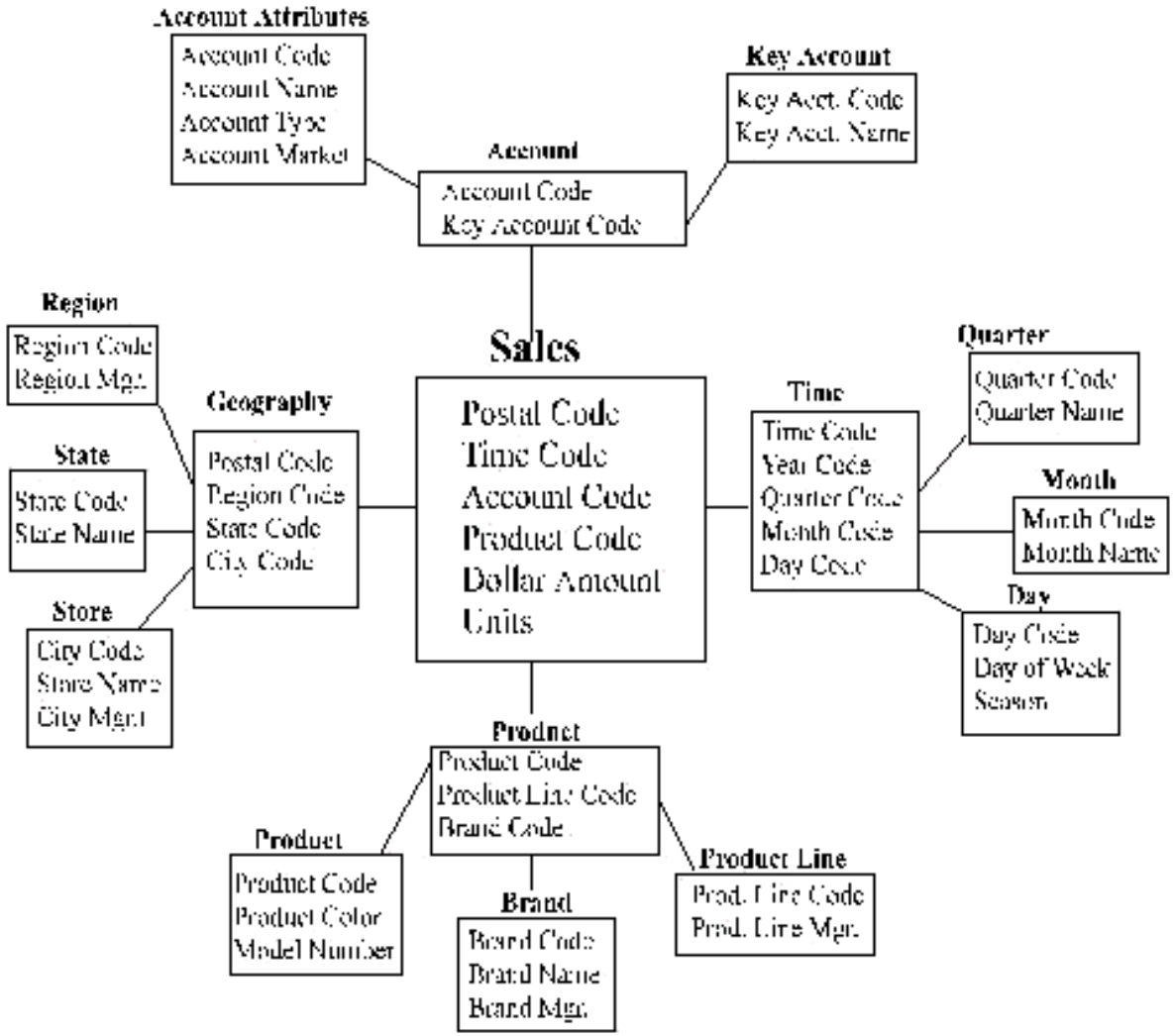
Схема “звезда”, пример



1

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

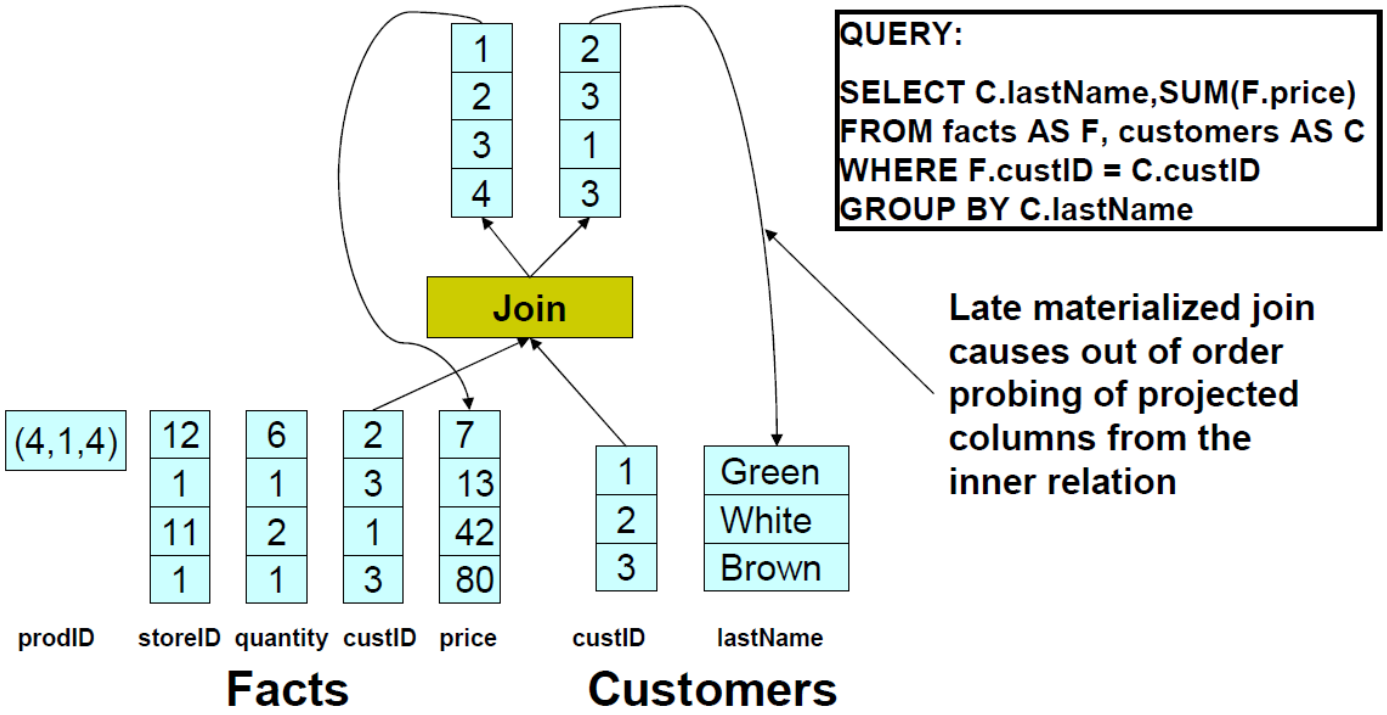
Схема “снежинка”, пример



[2]

[2] Изображение взято из http://www.informix.com.ua/articles/rolap/rolap.htm

Итоги прошлой лекции



[3]

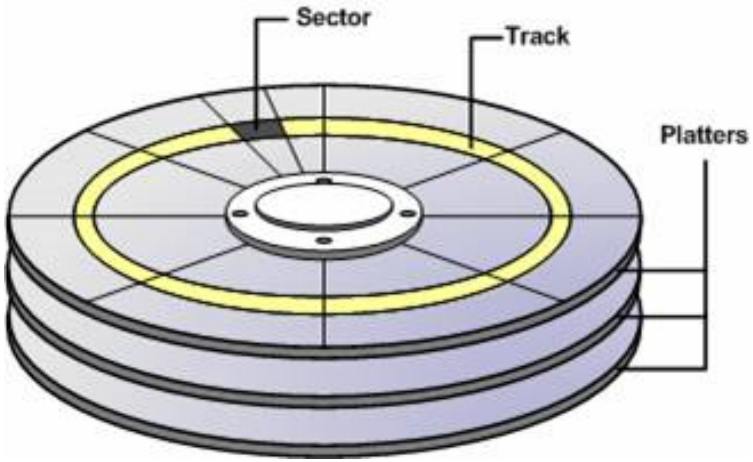
Соединения и поздняя материализация “не дружат”:

- По одной из колонок теряется отсортированность;
- Вычитка несортированной колонки очень дорога;

$\longrightarrow$ нужны другие операторы соединения, специализированные для этих нагрузок

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

Жесткий диск



[4]

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.
$\longrightarrow$ надо избегать случайного доступа

[4] Изображение взято из http://www.applexsoft.com/glossary/hard-disk.html

OLAP Benchmarks

$\longrightarrow$ нужны эффективные операторы соединения

Как? Использовать особенности OLAP, примеры:

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

Star Schema Benchmark

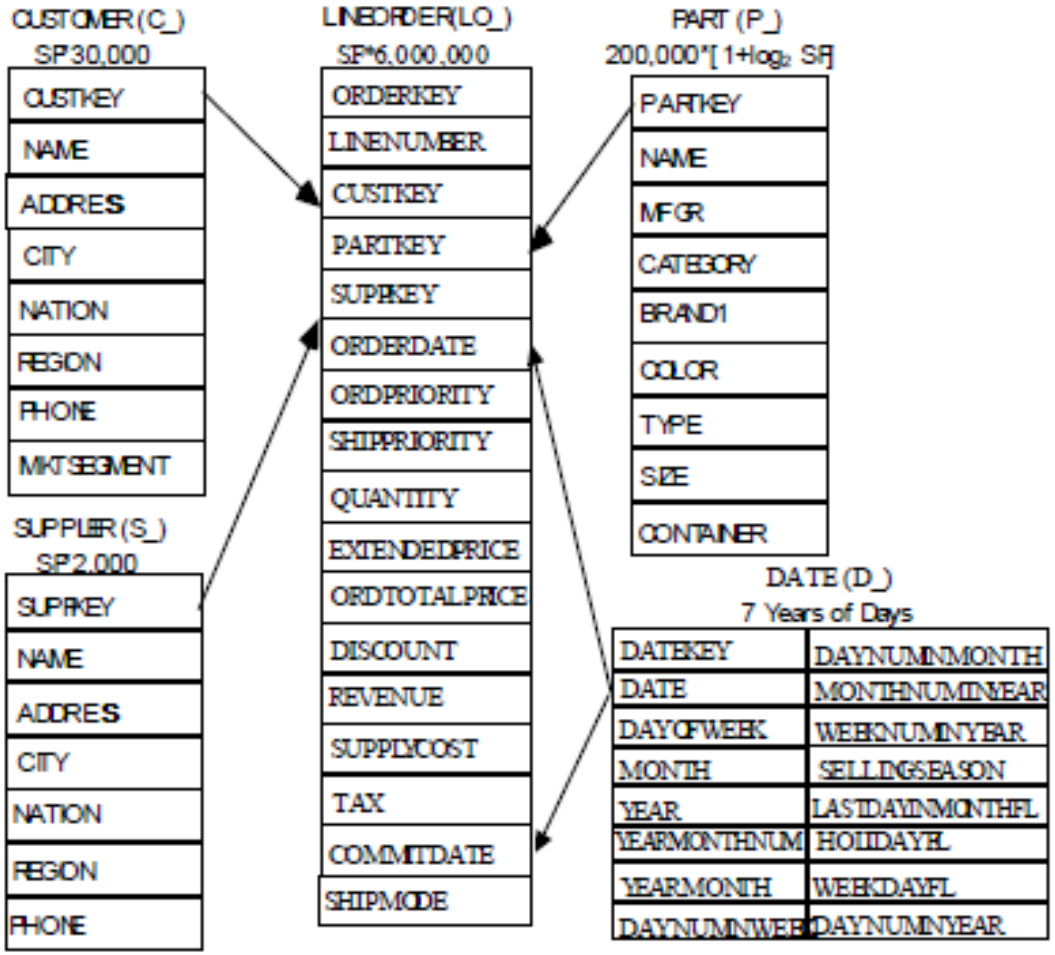


Figure 1.2 SSB Schema

5

[5] Изображение взято из [Star Schema Benchmark Specification, 2009]

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:

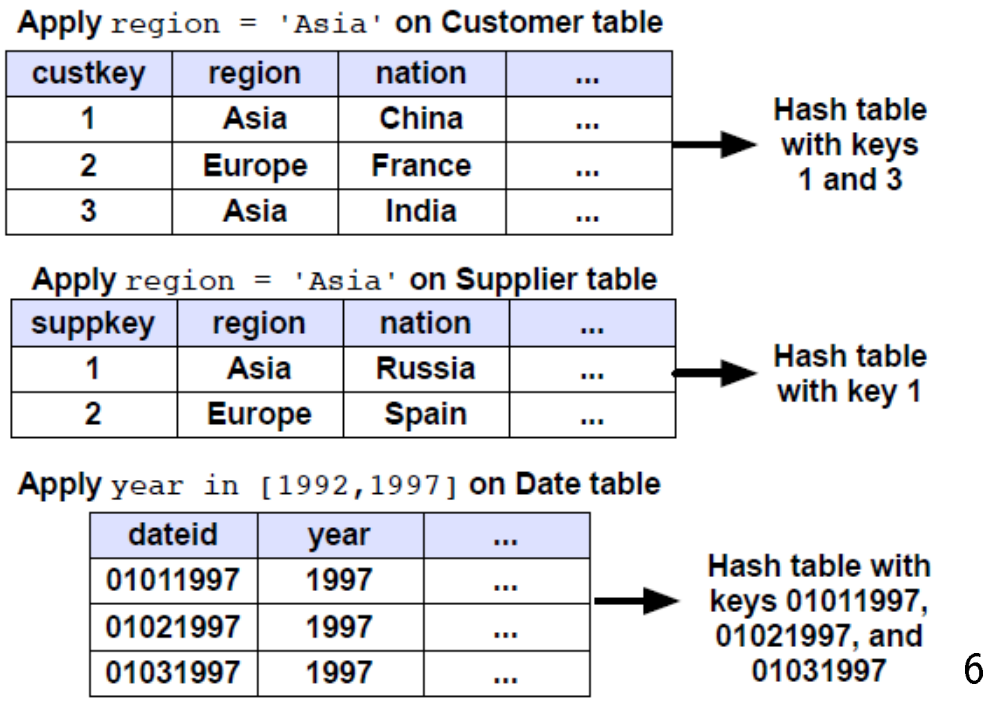
- Классический подход (ранняя материализация). Нет выигрышей от поздней материализации:
  - нельзя поэкономить на чтении с диска;
  - нет улучшения локальности кешей;
  - нет возможности оперировать над сжатыми данными.
- Поздняя материализация:
  - Значения в dimensions не отсортированы, придется вытаскивать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

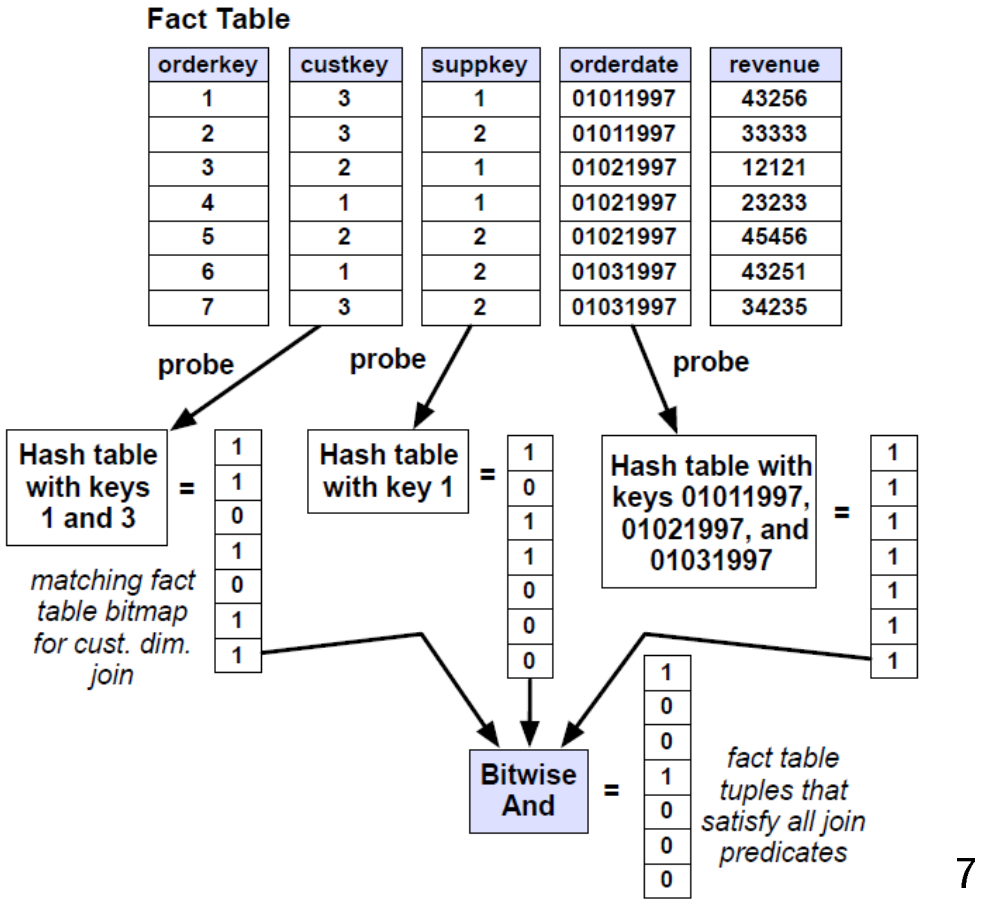


[6]

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

Invisible Join II

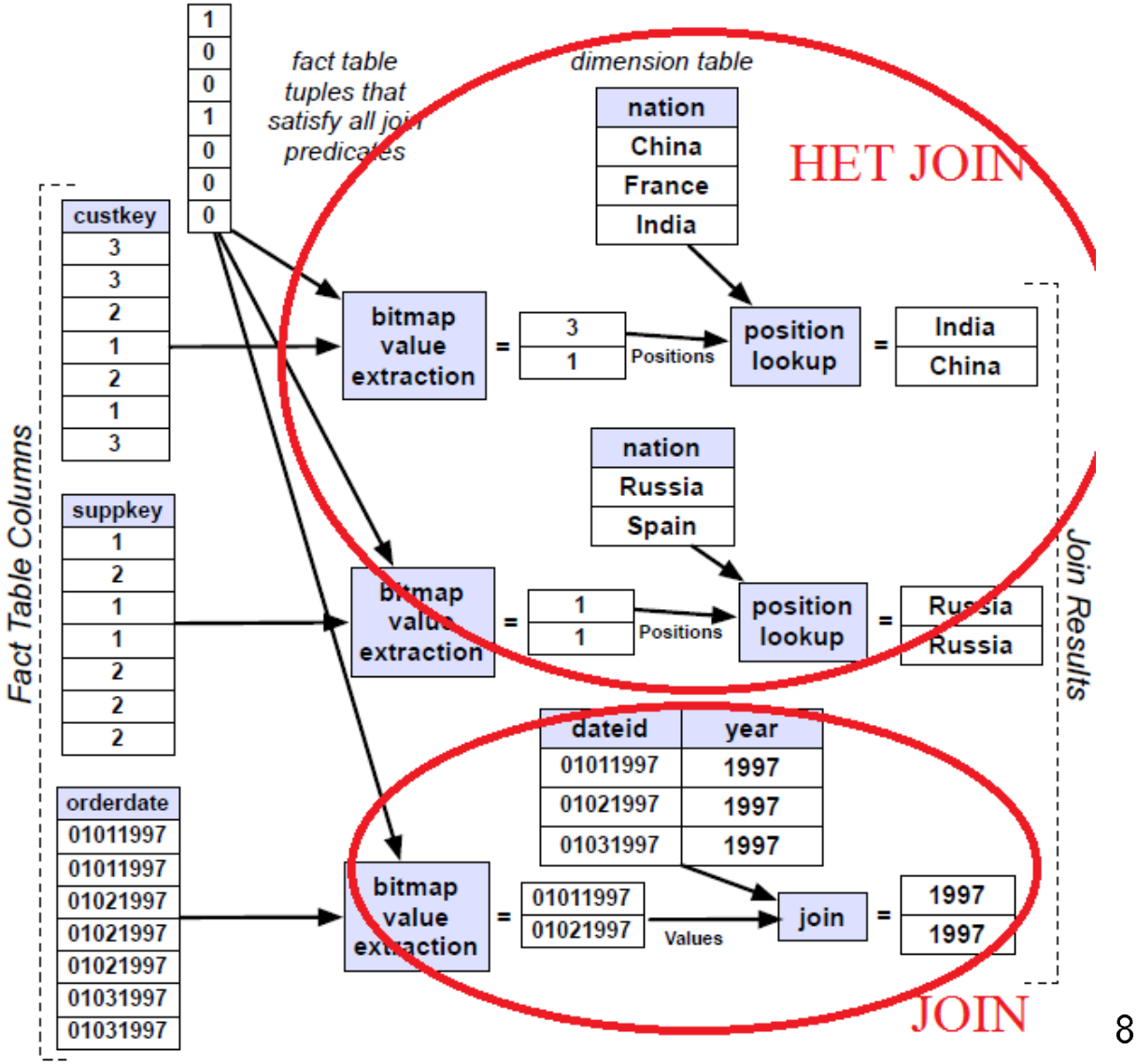
3. Идем по таблице фактов, опрашивая хеш-таблицы, от каждой получаем список позиций (записи удовлетворяющие предикату);
4. Полученные списки позиций пересекаем;



[7]

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

Invisible Join III (схема)



[8]

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

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 предыдщуем 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 выбирать меньше на этом шаге;

Оставляем таблицу фактов отсортированной всегда!

Jive join: описание

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).

Jive join I

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 I

Student	Course
Smith[1]	101
Smith[2]	109
Jones	104
Davis[1]	102
Davis[2]	105
Davis[3]	106
Brown	102
Black	103
Frick	107

Relation *Student*

Course	Instructor
101	Green
102	Yellow
103	Green
104	White
105	Evans
106	Alberts
106	Beige
108	Red
109	Grey

Relation *Course*

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

Join Result

Student tuple id	Course tuple id
1	1
2	9
3	4
4	2
5	5
6	6
6	7
7	2
8	3

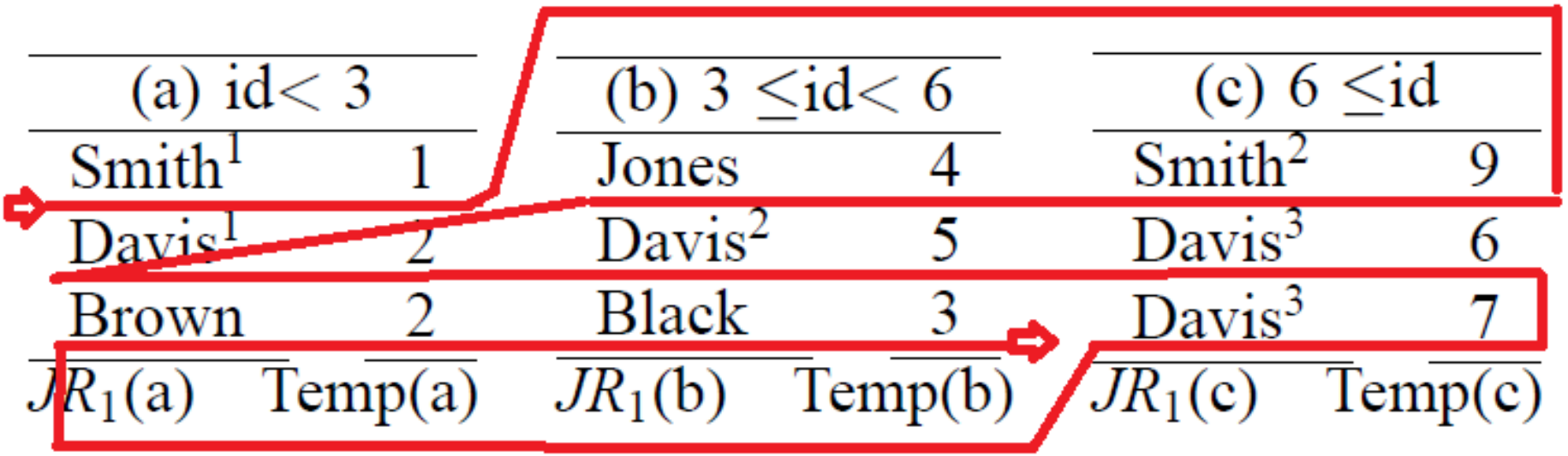
Join Index

9

[9] Изображение взято из [Li and Ross, 1999]

Jive Join пример II

Шаги J1 и J2:
Фрагменты: a) id < 3, b) 3 <= id < 6 и c) 6 <= id.



(a) id< 3		(b) 3 ≤id< 6		(c) 6 ≤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)	Temp(a)	JR_1(b)	Temp(b)	JR_1(c)	Temp(c)

10

Слева JR_1, справа temporary output file.

[10] Изображение взято из [Li and Ross, 1999]

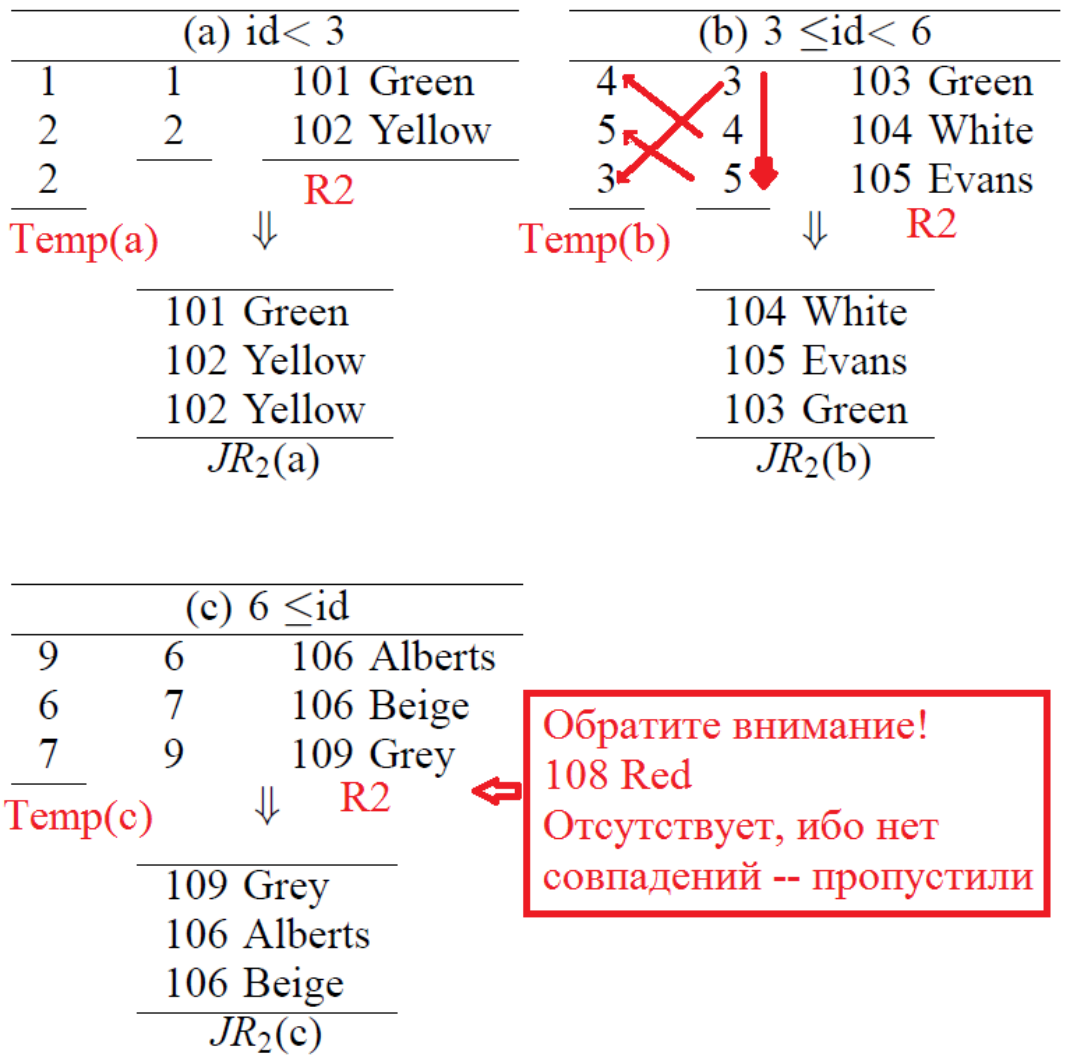
Jive join II

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;

Jive Join пример II

Шаг J3 (начало):



11

[11] Изображение взято из [Li and Ross, 1999]

Jive Join пример III

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

	Student
JR_1(a)	Smith[1]
	Davis [1]
	Brown
JR_1(b)	Jones
	Davis[2]
	Black
JR_1(c)	Smith[2]
	Davis[3]
	Davis[3]

Course	Instructor	
101	Green	
102	Yellow	JR_2(a)
102	Yellow	
104	White	
105	Evans	JR_2(b)
103	Green	
109	Grey	
106	Alberts	JR_2(c)
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:

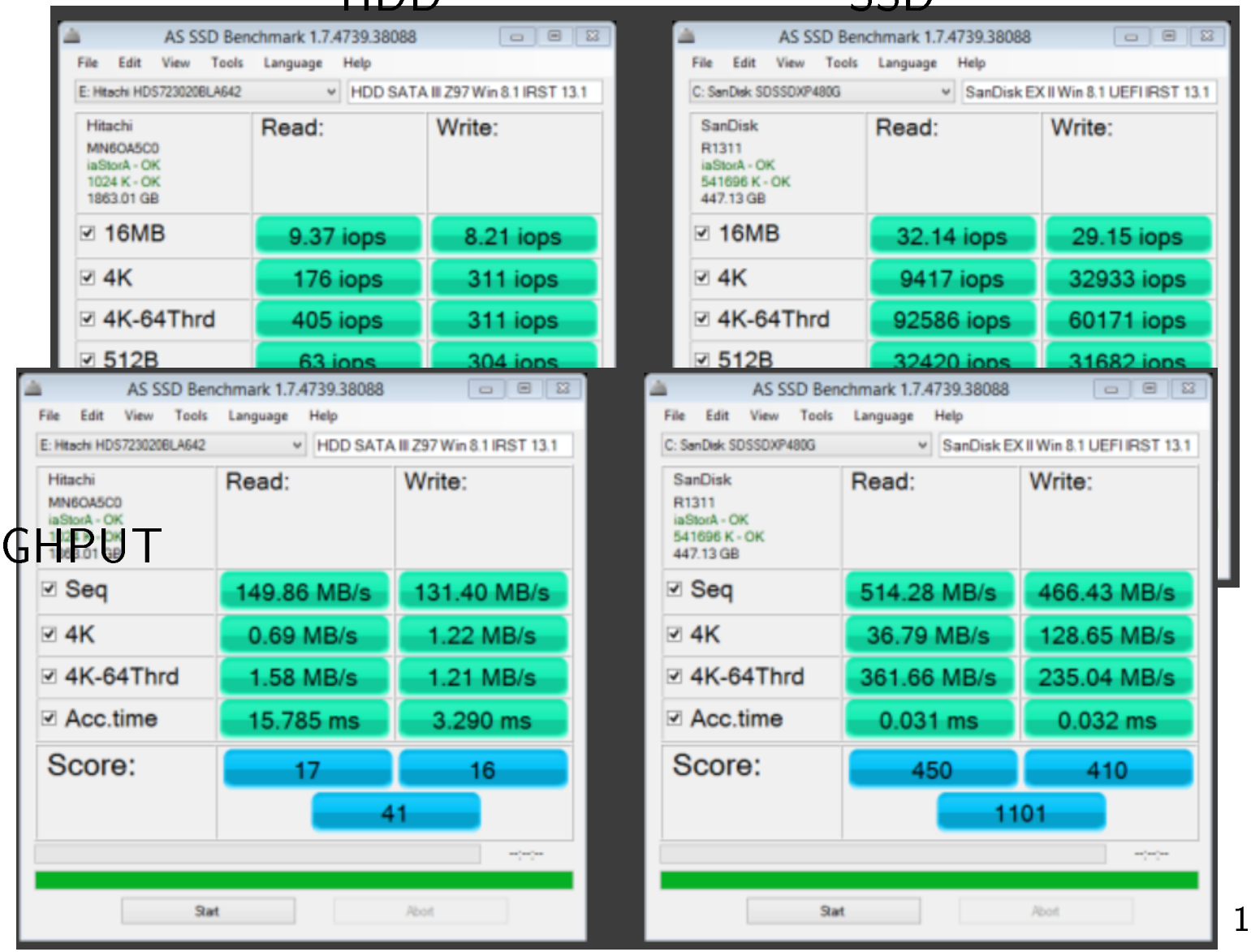
- Ломается pipelining, ждем конца J1 и J2;
- Нужен диск на промежуточные результаты — JR_1 + t. output files;
- Есть ограничения по оперативной памяти на R_2 (меньшее отношение), расчеты в статье;

Про диск II (latency, iops, throughput)

HDD SSD

IOPS

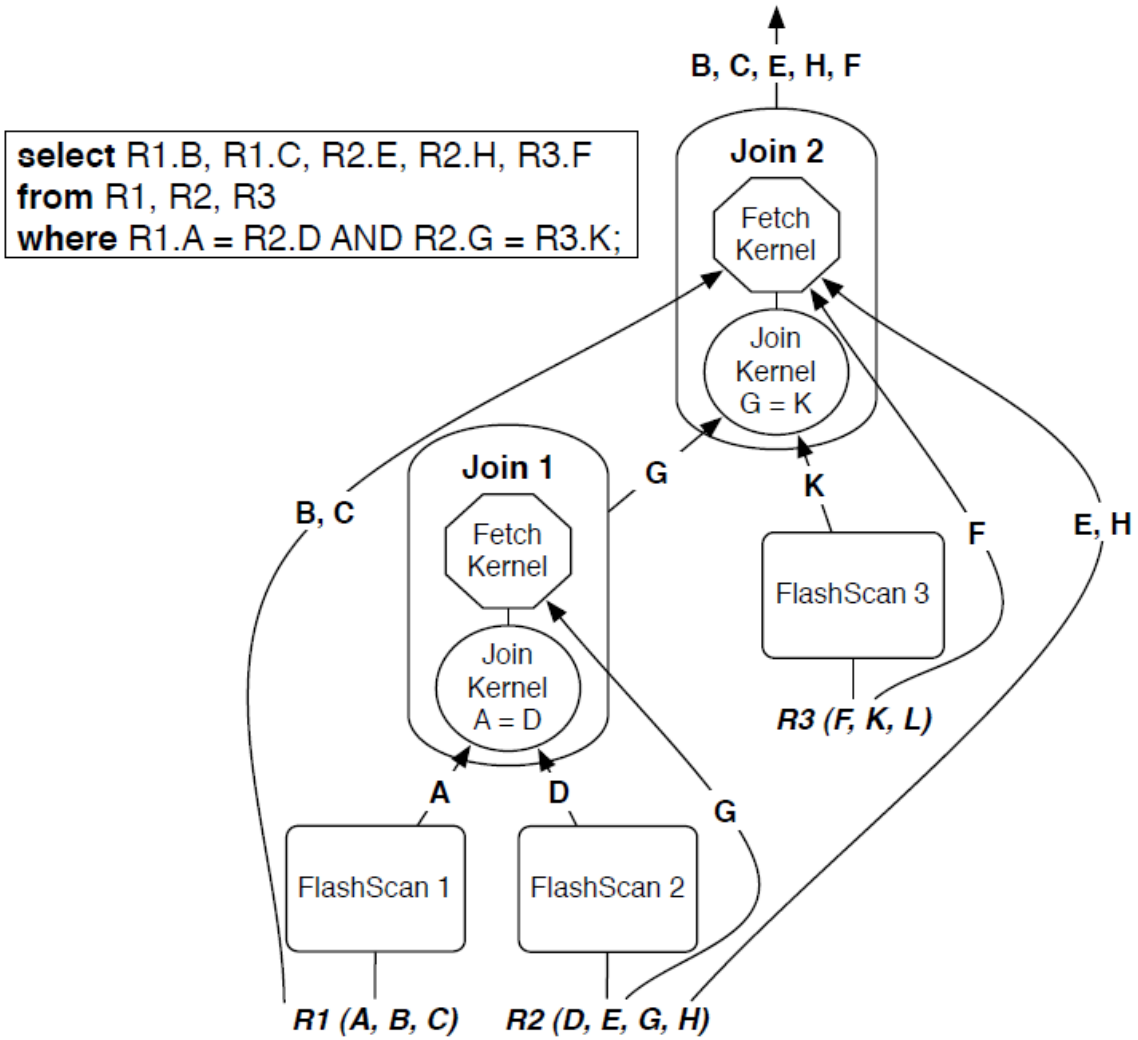
THROUGHPUT



13

[13] Изображения взяты из http://www.thessdreview.com/featured/ssd-throughput-latency-iopsexplained/

FlashJoin “на пальцах” I



14

[14] Изображение взято из [Tsirogiannis et al., 2009]

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;
- Использовав трюки наподобие Invisible Join, Jive Join, Flash Join, Radix Cluster/Decluster — 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-store[15]

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

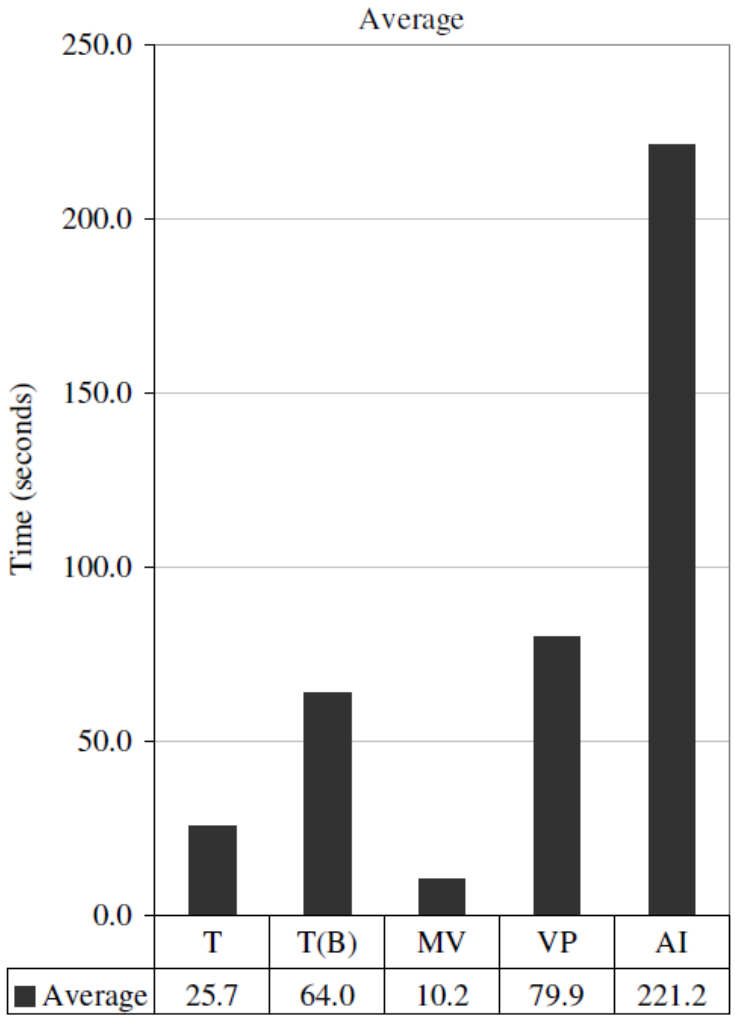
- VP: Полное вертикальное фрагментирование (ключ, атрибут);
  - Соединять с помощью HJ, пытались и кластеризованный индекс — хуже.
- AI: Index-only планы (каждый атрибут проиндексирован), так чтобы не читать таблицы вообще;
  - Дополнительный некластеризованный индекс.
- MV: Материализованное представление 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.

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

Сравнение



16

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

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

Какие особенности CS дают большее преимущество?

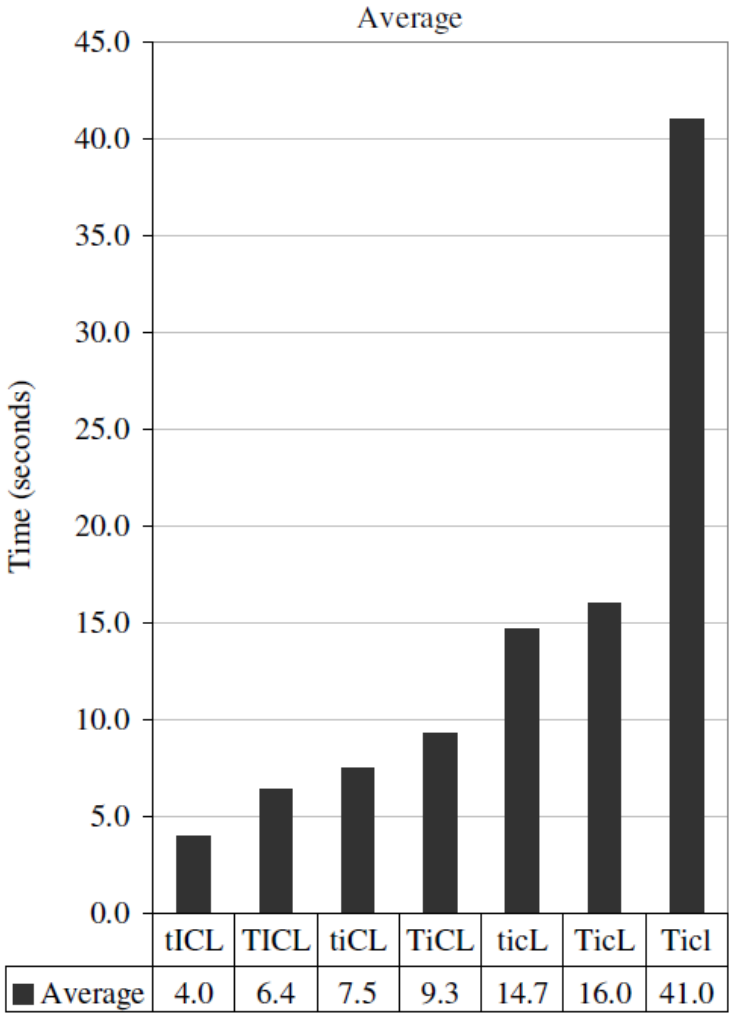
Особенности [Abadi et al., 2008]:

- 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

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

Изображения

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

Ссылки I

- 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

Ссылки II

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).