Принцип организации и преимущества «колонки-ориентированной» со сжатым словарем базы данных «в памяти»

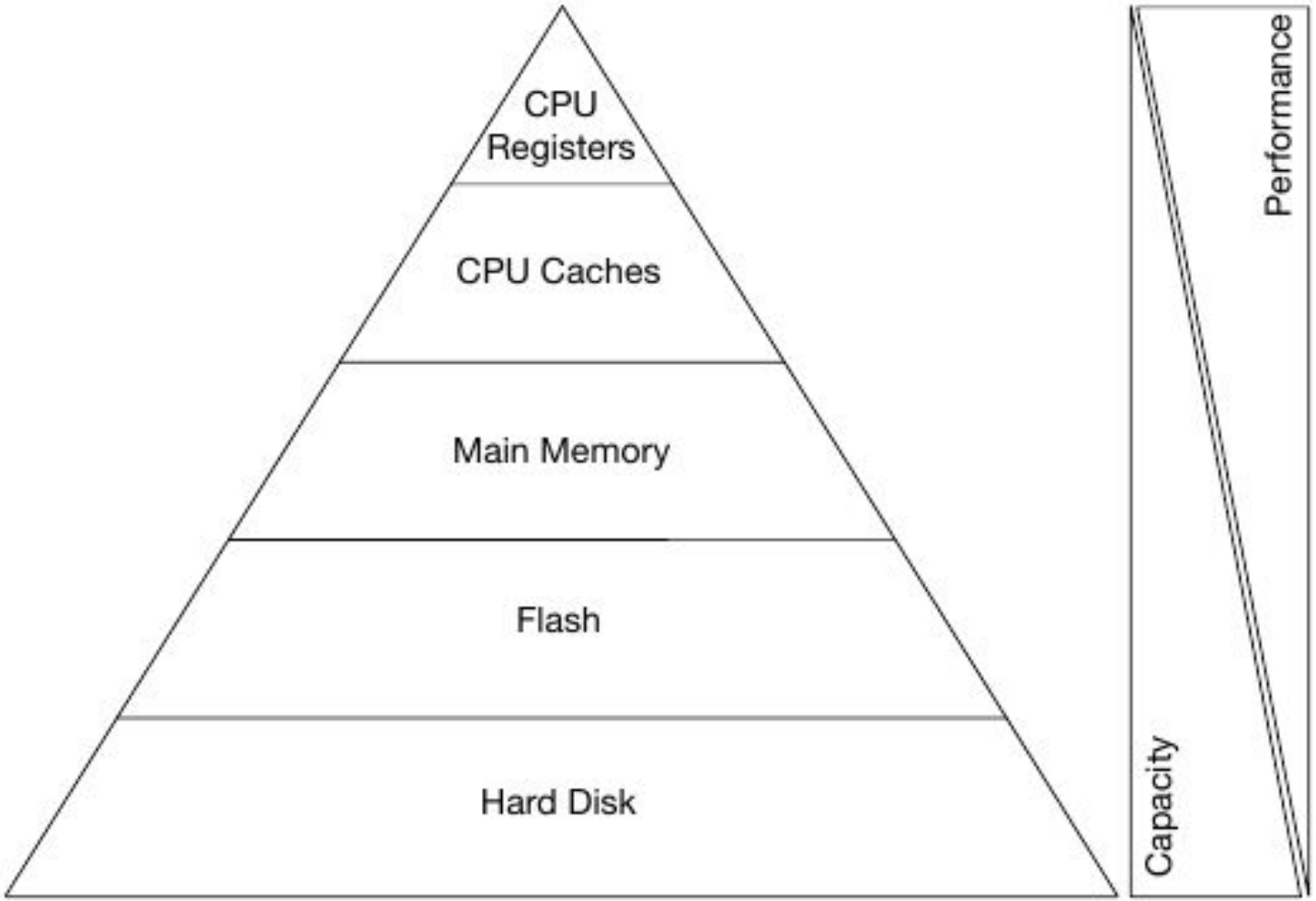
1. Изменения в аппаратном обеспечении

Прогресс в аппаратном обеспечении

-Мультиядерная архитектрура 8 x (8 – 15) ядер на каждый блейд-сервер;
- Параллельное масштабирование по всем блейд-серверам;
-Один блейд-сервер ≈ $ 50,000 = 1 сервер класса (масштаба) предприятия

-До 12 ТБ на современных серверных платах;
-- 85 Гб/с пропускная способность, CPU – DRAM;
--

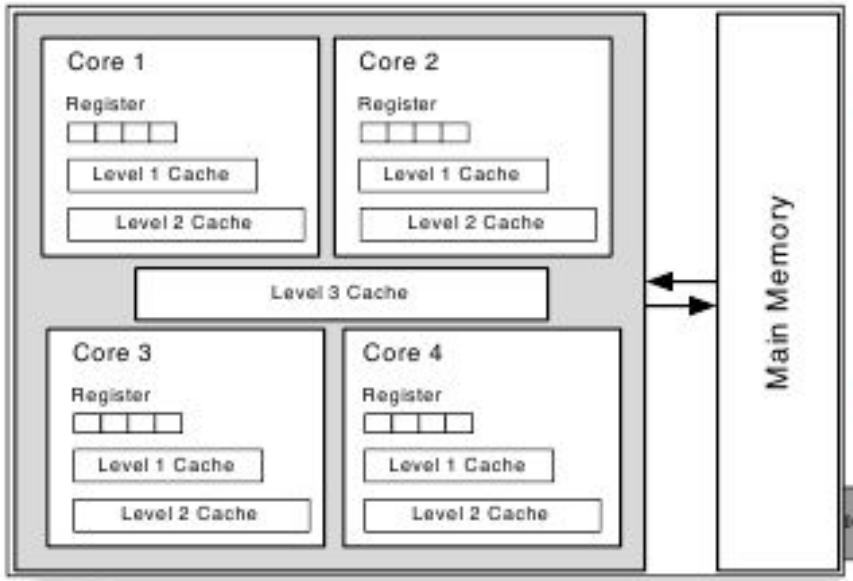
Иерархия памяти



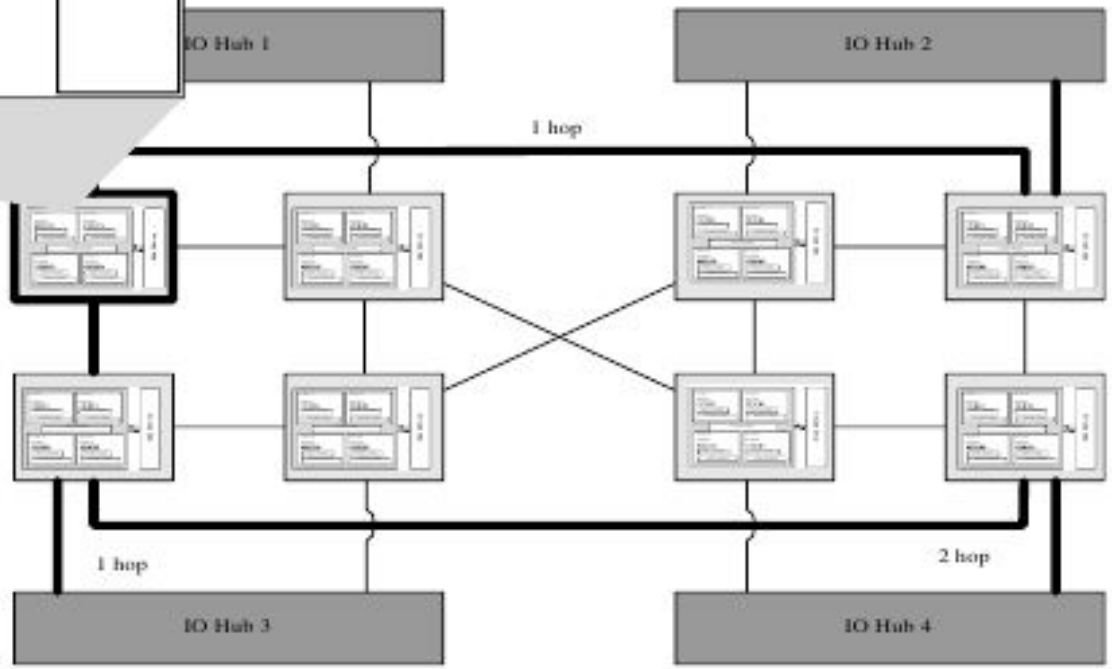
Показатели задержки

L1 cache reference (cached data word)	**0.5ns**	
Branch mispredict	5ns	
L2 cache reference	7ns	
Mutex lock/unlock	25ns	
Main memory reference	**100ns**	**0.1µs**
Send 2K bytes over 1 Gb/s network	20,000ns	20µs
SSD random read	150,000ns	150µs
Read 1 MB sequentially from memory	250,000ns	250µs
Disk seek	**10,000,000ns**	**10ms**
Send packet CA to Netherlands to CA	150,000,000ns	150ms

Архитектура CPU



- Non-Uniform Memory Access (NUMA) integrates many CPUs with many cores in a single system
- Not every memory access is local



В связи с тенденцией перехода в современных компьютерных системах от многоядерности к многоядерным системам и продолжением роста размера основной памяти, использование Front Side Bus (FSB) архитектуры с Единым доступом к амяти (Uniform Memory Access - UMA) стало узким местом в производительности и привело к тяжелым проблемам в проектировании аппаратных средств для подключения всех ядер и памяти.

Non-Uniform Memory Access (NUMA) пытается решить эту проблему путем введения местных участков памяти с дешевым доступом для соответствующих процессоров. Следующий рисунок показывает приближенно сравнение между архитектурными системами Единого доступа к памяти (UMA) и Non-Uniform Memory Access (NUMA).

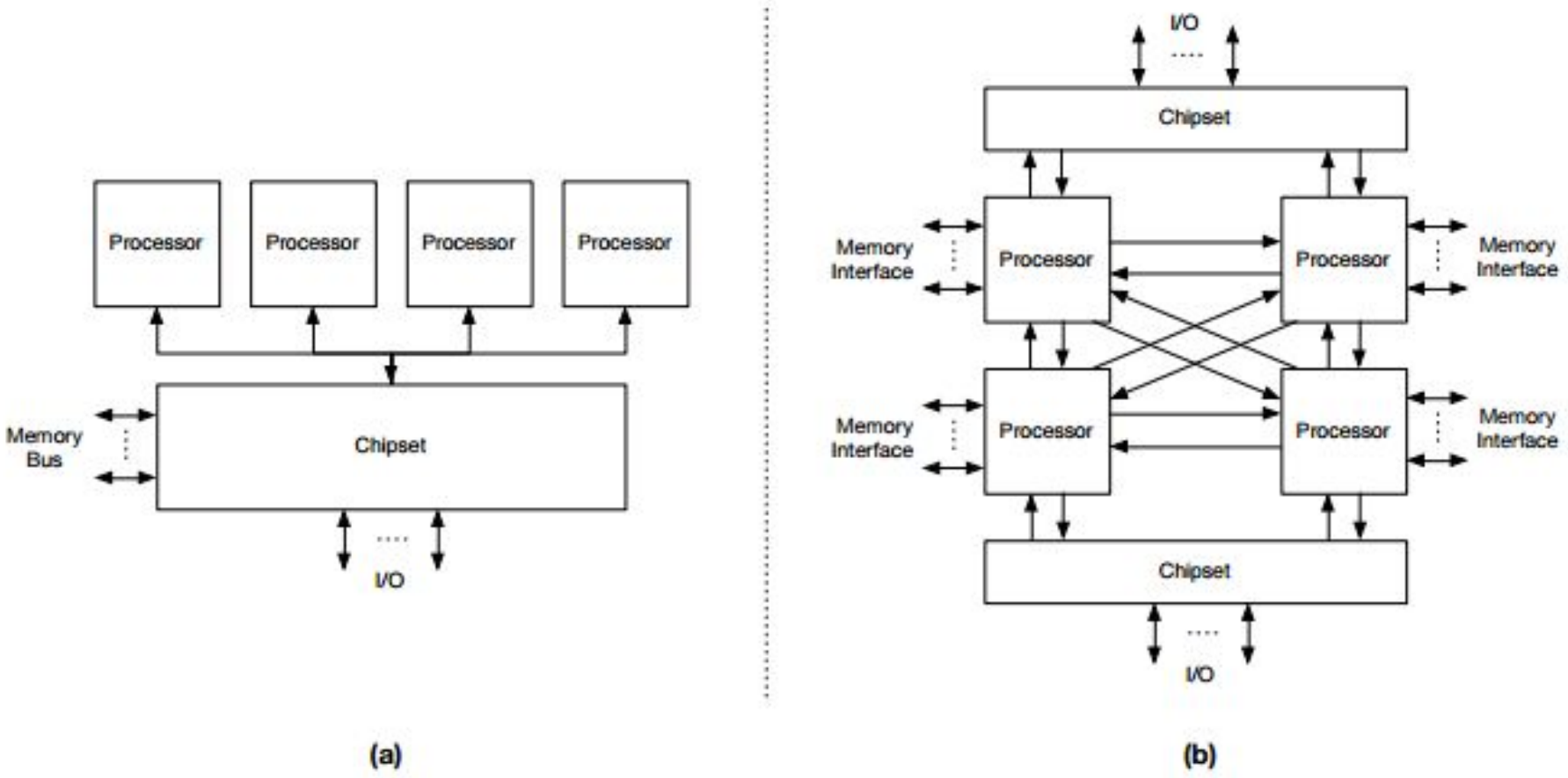


Fig. 4.4: (a) Shared FSB, (b) Intel Quick Path Interconnect [Int09]

Система UMA характеризуется детерминированным временем доступа для произвольного адреса памяти, независимо от того, какой процессор делает запрос, так как каждый чип памяти доступен через центральную шину памяти, как показано на рисунке 4.4 (а).

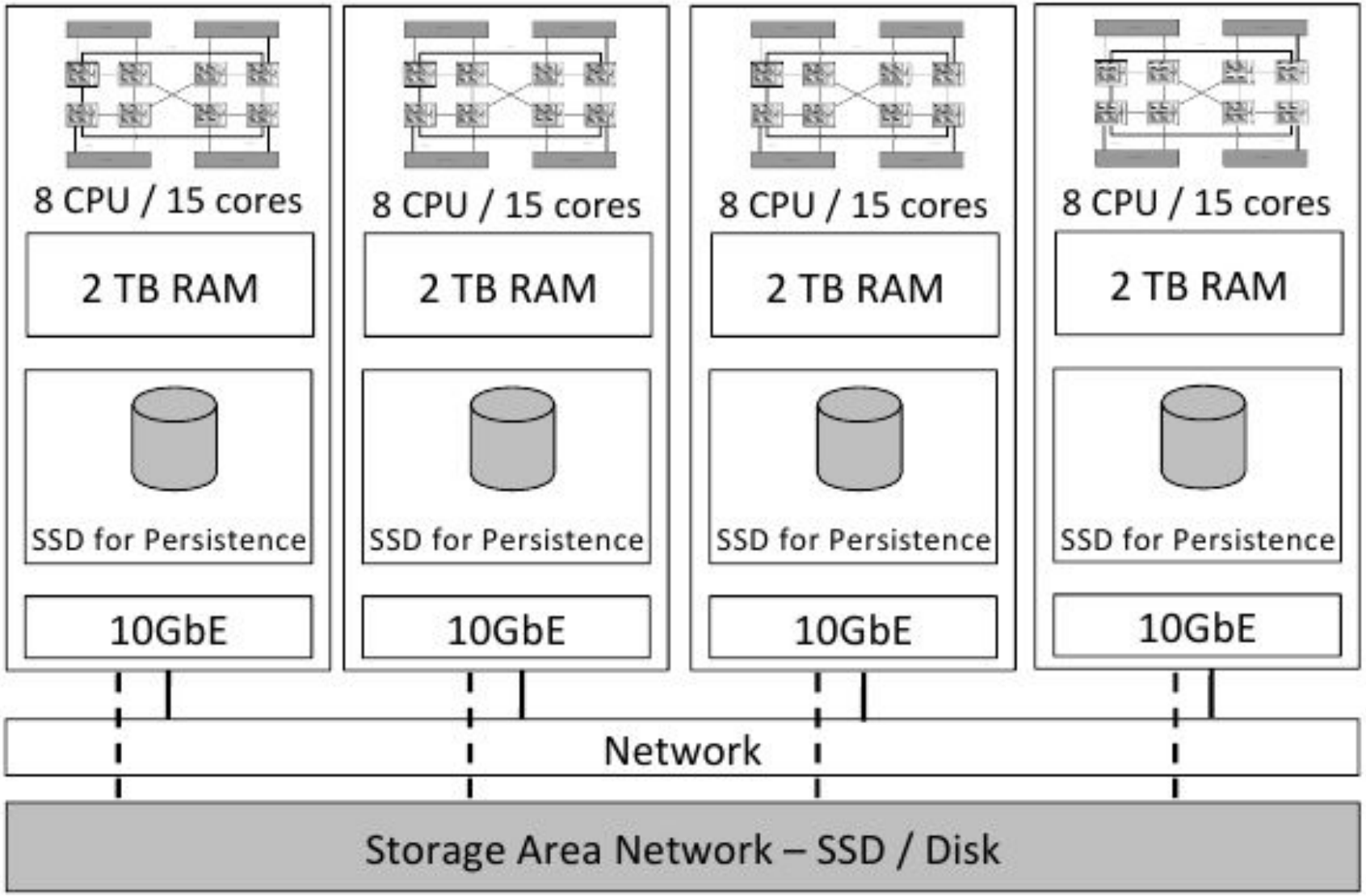
Для NUMA систем, с другой стороны, время доступа зависит от расположения памяти относительно процессора, например, местная (ближайшая) память может быть доступна быстрее, чем не-местная (рядом с другим процессором) память или разделяемая память (делится между процессорами), как показано на рисунке 4.4 (б).

Системы NUMA дополнительно классифицированы на кэш-когерентную NUMA (ccNUMA) и не кэш-когерентную NUMA. Системы ccNUMA обеспечивают каждому процессору кэш такой же, как для полной памяти и применяют когерентность с использованием протокола, реализованного на аппаратном уровне. Не кэш-когерентная система NUMA требует программный уровень для соответствующей обработки конфликтов памяти.

Хотя не ccNUMA аппаратно проще и дешевле построить, большинство из сегодняшнего доступного стандартного аппаратного обеспечения ccNUMA, так как не ccNUMA труднее программировать.

Чтобы в полной мере использовать потенциал NUMA, приложения должны быть осведомлены о различных местоположениях памяти и должны в первую очередь загружать данные из локальной памяти процессора. Приложения с привязкой к памяти могут испытывать ухудшение до 25% от их максимальной производительности, если обращаются к нелокальной памяти вместо локальной.

Масштабирование основной памяти системы



Следующий рисунок показывает пример установки для масштабирования системы управления базами данных на основе основной памяти с использованием нескольких узлов (горизонтально масштабируемое устройство).

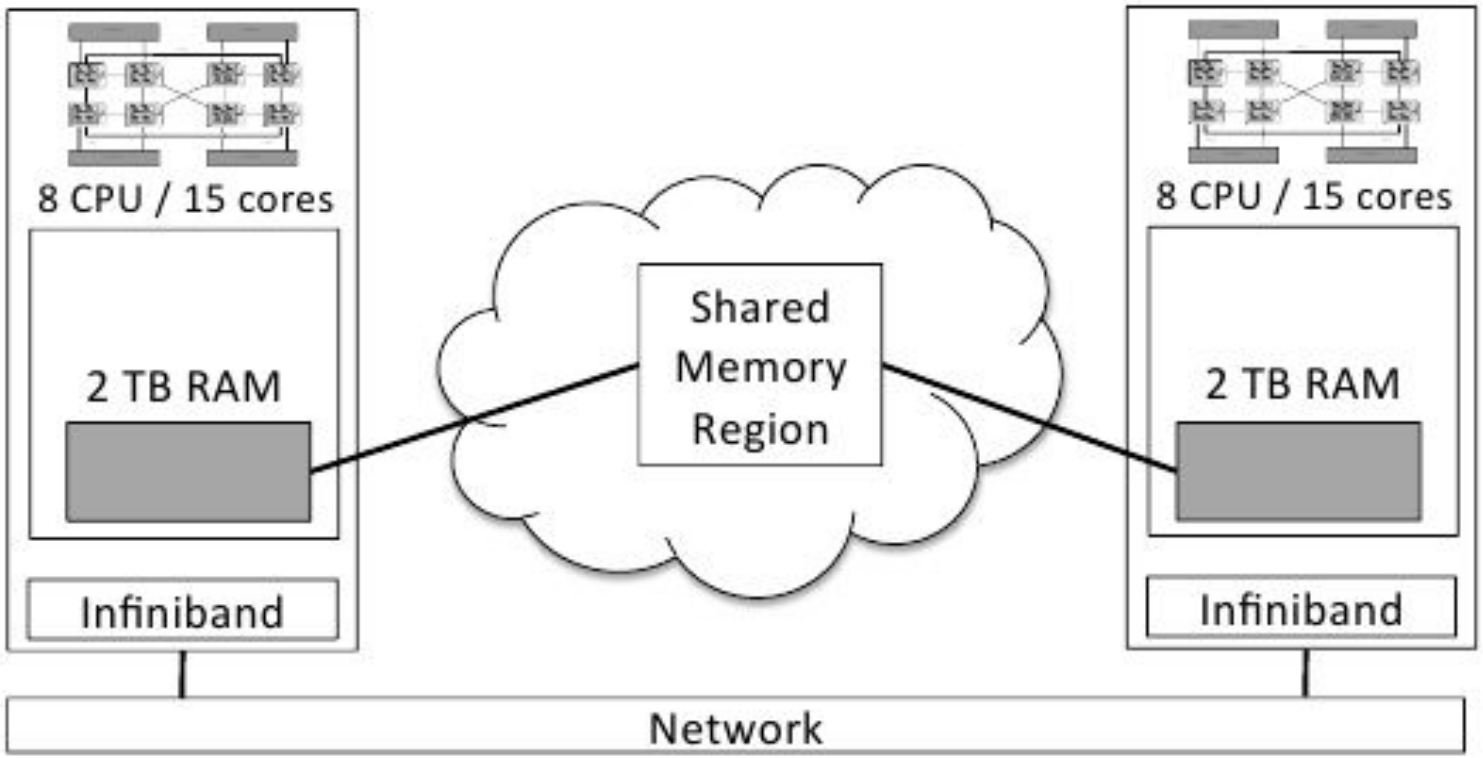
Каждый узел содержит восемь процессоров с пятнадцатью ядрами, составляя в целом 120 ядер на одном узле и 480 ядер на четырех узлах. С каждым узлом, содержащим два терабайта ОЗУ, система может хранить в общей сложности восемь терабайт данных полностью в памяти. Используя твердотельные накопители SSD или традиционные диски для постоянных операций, таких как ведение логов, архивирование и аварийное восстановление для данных, система может легко перегрузить данные в памяти в случае планового или внепланового перезапуска одного или нескольких узлов.

Удаленный прямой доступ к памяти

Использование разделяемой памяти для прямого доступа к памяти на удаленных узлах все больше становится альтернативой традиционным сетевых коммуникаций.

Для узлов, соединенных через канал InfiniBand, можно устанавливать общую область памяти для автоматического доступа к данным на различных узлах без явного запроса данных от соответствующего узла. Это прямой доступ без необходимости транспортировки и обработки на удаленном узле очень легкий способ масштабирования по вертикали вычислительную мощность. Исследования, проведенные в Стэнфордском университете в сотрудничестве с HPI помощью масштабную RAM кластер показывает очень обнадеживающие результаты, таким образом, предлагая прямой доступ к кажущемуся неограниченным количеству памяти с минимальными накладными расходами.

Удаленный прямой доступ к памяти

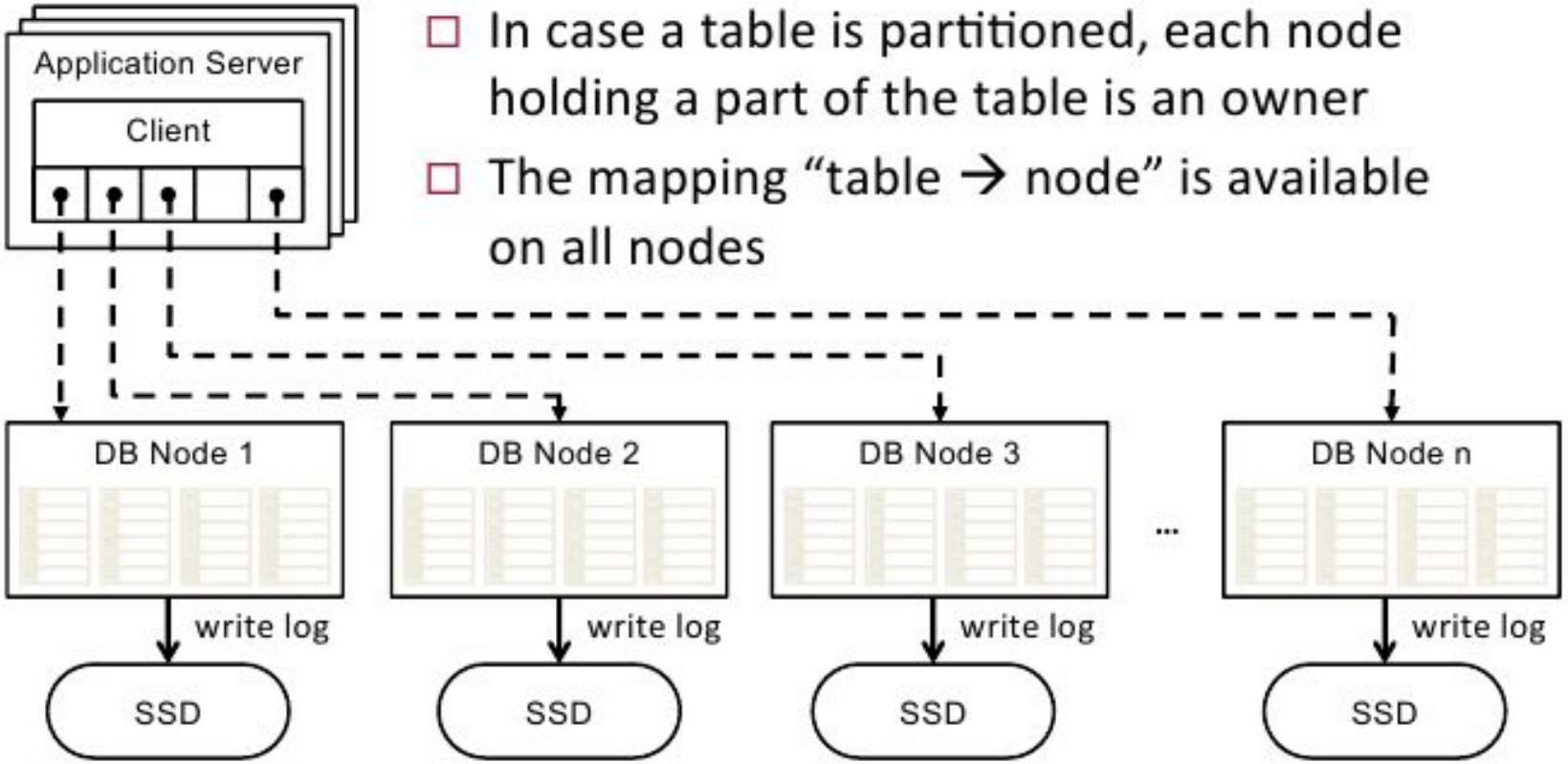


- **InfiniBand** (последовательная, коммутируемая сетевая архитектура) – позволяет снизить задержку коммутации;
- RDMA под Infiniband позволяет заменить традиционные SAN в качестве носителя информации,

Распределенное соединение

Tables are distributed across nodes:

- Each table is owned by one node in the cluster
- In case a table is partitioned, each node holding a part of the table is an owner
- The mapping “table → node” is available on all nodes



Выводы

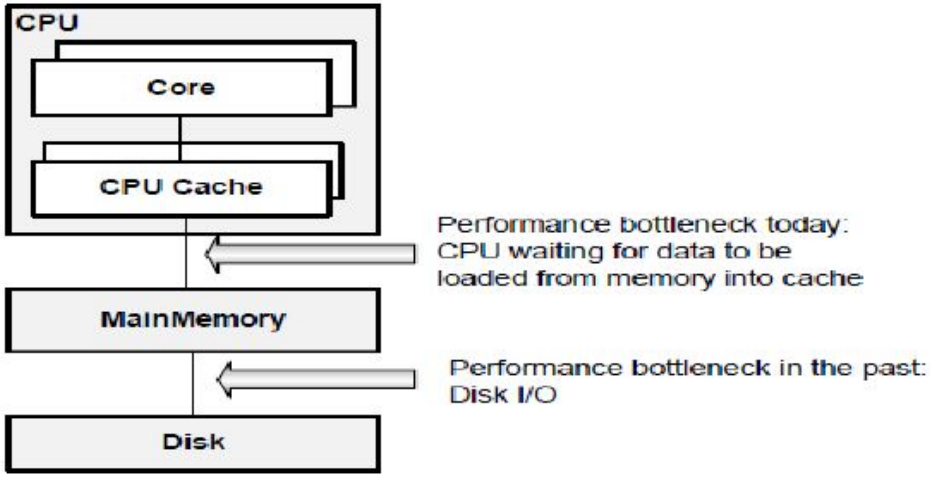
1. Multi-Core и In-Memory
2. Новое «бутылочное горлышко» между ЦПУ кэш и ОЗУ

Традиционные базы данных

- Узкое место дисковые операции чтения/записи в связи с ограничением оперативной памяти

HANA

- Благодаря многоядерной архитектуре и хранению данных в памяти единственное узкое место кэш CPU и оперативная память



2. Кодирование словаря

Так как память является новым узким местом, требуется минимизировать доступ к ней. Доступ к меньшему количеству столбцов, лишь на выполнение запросов на необходимые атрибуты (свойства объектов в БД) может сделать это с одной стороны. С другой стороны, уменьшение числа битов, используемых для представления данных, может снизить как потребление памяти, так и время передачи.

Кодирование словаря создает основу для ряда других методов сжатия, которые могут быть применены сверх кодирования столбцов. Основным эффектом кодирования словаря является то, что длинные значения, такие как тексты, будут представлены в виде коротких целочисленных значений.

Эффективное хранение данных

Традиционные СУБД

- Даже когда данные в оперативной памяти, CPU тратит ½ времени выполнения на догрузку (ожидание доставки данных из RAM в кэш CPU) или обработку отсутствия данных в кэше

HANA

- Эффективное кэширование данных – минимизация доставки данных в кэш CPU
- Ускорение поиска необходимых данных за счет поколоночного хранения

Table

Country	Product	Sales
US	Alpha	3.000
US	Beta	1.250
JP	Alpha	700
UK	Alpha	450

Row Store

Row 1	US
	Alpha
	3.000
Row 2	US
	Beta
	1.250
Row 3	JP
	Alpha
	700
Row 4	UK
	Alpha
	450

Column Store

Country	US
	US
	JP
	UK
Product	Alpha
	Beta
	Alpha
	Alpha
Sales	3.000
	1.250
	700
	450

SanssouciDB является прототипом системы баз данных для унифицированной аналитической и транзакционной обработки. Концепция SanssouciDB построена на прототипах, разработанных HPI и на существующей системе баз данных фирмы SAP. SanssouciDB является базой данных SQL и содержит такие же компоненты, как и другие базы данных, такие как построитель запросов, исполнитель (мастер отчетов), мета-данные, менеджер транзакций и т.д.

SanssouciDB реализует несколько основополагающих для «in memory» новых принципов.

Данные хранятся в основной памяти

В отличие от большинства других баз данных, данные SanssouciDB постоянно держит в основной памяти.

Оперативная память в ней является основным местом сохранения для данных, но для хранения истории (логов) и информации для восстановления по-прежнему необходим диск долговременного хранения данных. Все операции, например, поиск, объединение, или агрегация, могут предполагать, что данные находятся в оперативной памяти. Таким образом, операторы могут быть запрограммированы по-другому, и освобождаются от любых проблем, возникающих в процессе оптимизации доступа к диску. Использование оперативной памяти в качестве основной прямо ведет к появлению другого принципа организации данных. Принцип - работа с данными только

База данных «столбец-ориентированная»

Другая концепция, используемая в SanssouciDB, была изобретена более чем два десятилетия назад: хранение данных по столбцам, а не построчно. В столбец-ориентации, полные столбцы хранятся в соседних блоках. Это можно сравнить со строчно-ориентированным хранением, где полные записи (строки) хранятся в соседних блоках. Столбец-ориентированное хранение, в отличие от строчно- ориентированного, хорошо подходит для чтения последовательных записей из одного столбца. Это может быть полезно для агрегации и сканирования

Чтобы свести к минимуму количество данных, которые должны быть переданы между местом их хранения и процессором, SanssouciDB использует несколько различных методов сжатия данных.

Следствия столбец-ориентации

Столбец-ориентированное хранение получило широкое распространение в системах баз данных, специально разработанных для OLAP, где преимущество столбец-ориентированного хранения очевидно в случае квази-последовательного сканирования отдельных атрибутов и дальнейшей их обработки. Если не все полях таблицы запрашиваются, столбец-ориентация может быть использована также хорошо в обработке транзакций (избегая "SELECT *").

Анализ корпоративных приложений показал, что на самом деле нет ни одно приложения, которое бы использовало все поля записи. Например, в «Напоминаниях» только 17 атрибутов из таблицы, содержащей 300 атрибутов, необходимы. Если только 17 атрибутов запрашиваются вместо 300, возникает мгновенное преимущество во времени сканирования данных, может быть достигнуто.

Кодирование словаря является относительно простым. Это означает не только то, что его легко понять, но также, что его легко осуществить и нет нужды в сложных многоуровневых процедурах, которые могли бы ограничить или уменьшить прирост производительности. Во-первых, рассмотрим пример, представленный на рис, для разъяснения в общем виде алгоритм перевода исходных значений в целые числа.

Пример кодирования словаря и сжатия столбцов по таблице населения Земли

- 8 billion humans

- Attributes:
  - first name
  - last name
  - gender
  - country
  - city
  - birthday

  → 200 byte per tuple

- Each attribute is dictionary encoded

Table: world_population

recID	fname	lname	gender	city	country	birthday
...	...	...	...	...	...	...
39	John	Smith	m	Chicago	USA	12.03.1964
40	Mary	Brown	f	London	UK	12.05.1964
41	Jane	Doe	f	Palo Alto	USA	23.04.1976
42	John	Doe	m	Palo Alto	USA	17.06.1952
43	Peter	Schmidt	m	Potsdam	GER	11.11.1975
...	...	...	...		...	...

Кодирование словаря по столбцам

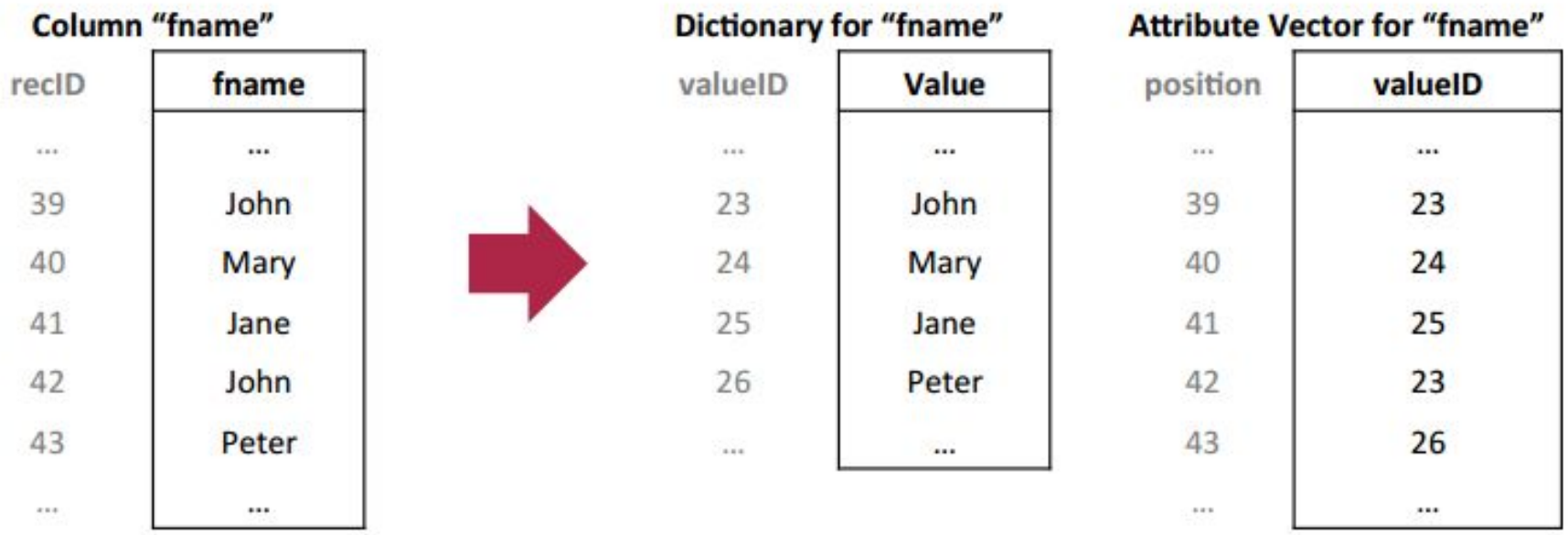
¨ столбец разделен на словарь и вектор атрибутов

¨ словарь сохраняет все уникальные значения с неявной valueID

¨ Атрибут векторные магазины valueIDs для всех записей в столбце

¨ Позиция хранится неявно

¨ Позволяет сместить разрядную кодировку типов данных фиксированной длины



Кодирование словаря применяется к каждому столбцу таблицы отдельно. В примере, каждое отличающееся значение имени в первом столбце «имя» заменяется отдельным целочисленным значением. Положение текстового значения (например, Мэри) в словаре также представляется целым номером ("24" для Мэри).
До сих пор мы не сэкономили место для хранения. Преимущества приходят, когда значения появляются более, чем один раз в столбце. В нашем маленьком примере, значение "Джон" можно найти два раза в колонке "имя", а именно на позициях 39 и 42. С использованием словаря кодирования, длинные текстовые значения (мы предполагаем, 49 байт на запись в первом столбце имени) представлены коротким целочисленным значением (23 бита необходимо для кодирования 5 миллионов различных имен, существующих в мире). Появятся чаще одинаковые значения, тем лучше: кодированный словарь может позволить сжать колонку. Как мы уже отмечали, данные предприятий имеют низкую энтропию. Таким образом, кодированный словарь хорошо подходит и дает хороший коэффициент сжатия в таких случаях. Далее, мы вычислим возможную экономию памяти для первых имен и гендерных колонок в нашем примере населения мира .

Принимая во внимание таблицу населения Земли на 8 миллиардов строк и 200 байт на строку:

Attribute	# of Distinct Values	Size
first name	5 million	49 Byte
last name	8 million	50 Byte
gender	2	1 Byte
country	200	49 Byte
city	1 million	49 Byte
birthday	40 000	2 Byte
	Sum	200 Byte

полный объем данных:

8 миллиардов строк · 200 байт для каждой строки = 1,6 Тб

Сколько битов потребуется для представления всех 5 миллионов различных значений первого столбца имя "имя"?

$$\lceil log_2(5{,}000{,}000) \rceil = 23$$

Таким образом, 23 бита достаточно, чтобы представлять все различные значения для требуемого столбца. Вместо того, чтобы использовать

8 млрд · 49 байт = 392 000 000 000 байт = 365.1 ГБ

для первого столбца имен, сам вектор атрибутов может быть уменьшен до размера

8 млрд · 23 бит = 184 млрд бит = 23 млрд байт = 21.4 ГБ

и вводится дополнительный словарь, который должен содержать

49 байт · 5 миллионов = 245 млн байт = 0,23 ГБ.

Достигнутый коэффициент сжатия может быть рассчитан следующим образом:

$$\frac{uncompressed\ size}{compressed\ size} = \frac{365.1\,GB}{21.4\,GB + 0.23\,GB} \approx 17$$

Это означает, что мы сократили размер столбца в 17 раз и в результате структуры данных потребляют только около 6 % от исходного количества оперативной памяти.

Пример кодирования словаря: пол

Теперь, давайте посмотрим на столбец пол. Он имеет лишь 2 различных значения. Для того, чтобы представлять пол («м» или «ж») без сжатия для каждого значения, требуется 1 байт. Так, без сжатия, объем данных составляет:

8 млрд · 1 байт = 7,45 ГБ

Если используется сжатие, то 1 бита достаточно, чтобы представить ту же информацию.

Вектор атрибута требует:

8 млрд · 1 бит = 8 млрд бит = 0.93 ГБ

пространства. Словарь , необходимый дополнительно:

Достигнутый коэффициент сжатия может быть рассчитан следующим образом:

$$\frac{uncompressed\ size}{compressed\ size} = \frac{7.45GB}{0.93\,GB + 2\,Byte} \approx 8$$

Степень сжатия зависит от размера первоначального типа данных, а также по энтропии столбца, которая определяется двумя мощностями (кардинальностями):

- Колонка-кардинальность: число различных значений в столбце
- Таблица-кардинальность: общее количество строк в таблице или столбце

Энтропия является мерой, которая выражает, как много содержится информации в колонке (мера плотности информации).

Она рассчитывается следующим образом:

Энтропия = столбцовая кардинальность / табличная кардинальность

Column	Cardi-nality	Bits Needed	Item Size	Plain Size	Size with Dictionary (Dictionary + Column)	Compression Factor
First names	5 millions	23 bit	50 Byte	400GB	250MB + 23GB	≈ 17
Last names	8 millions	23 bit	50 Byte	400GB	400MB + 23GB	≈ 17
Gender	2	1 bit	1 Byte	8GB	2b + 1GB	≈ 8
City	1 million	20 bit	50 Byte	400GB	50MB + 20GB	≈ 20
Country	200	8 bit	47 Byte	376GB	9.4kB + 8GB	≈ 47
Birthday	40000	16 bit	2 Byte	16GB	80kB + 16GB	≈ 1
Totals			**200 Byte**	**≈ 1.6TB**	**≈ 92GB**	**≈ 17**

Отсортированный словарь

Записи словаря сортируются либо по их числовым значением или лексикографически

сложность словаря тогда выглядит так:

O(log(n)) вместо O(n)

Критерии отбора тогда будут иметь меньший диапазон

Записи словаря могут быть дополнительно сжаты, чтобы уменьшить объем необходимого хранения

Преимущества кодированного словаря увеличиваются, если к словарю применяется сортировка.

Получение значения из отсортированного словаря ускоряет процесс поиска от O (N), что означает полное сканирование словаря, к O (log(п)), потому что значение в словаре можно найти, используя бинарный поиск.

Но эта оптимизация имеет свою цену. Словарь должен быть повторно отсортирован каждый раз, когда добавляется новое значение, которое не принадлежит к концу отсортированной последовательности. В этом случае позиции уже присутствующих значений, которые находятся за вставленным значением, должны быть отброшены на одну позицию. Если сортировка словаря ничего не будет стоить, обновление связанного вектора атрибута напротив, будет. В нашем примере, около 8 млрд значений должны быть проверены или обновлены, если, например, новое имя добавляется в словарь.

Выводы

Преимущества колоночного хранения

- Высокий уровень компрессии данных при их избыточности (низкой кардинальности)
  Примеры: run-length encoding, cluster coding, dictionary coding
- Высокая производительность колоночных операций
  Примеры: поиск/агрегация циклами по массивам последовательно хранящихся данных
- Отсутствие необходимости в дополнительных индексах
  Хранение поколоночно – хранение данных в виде индекса
- Параллелизация
  Колоночное хранение уже вертикально партиционировано, поэтому операции по разным полям таблицы легко выполнить параллельно
- Отсутствие необходимости в материализации данных
  Поле таблицы в оперативной памяти позволяет выполнить агрегацию на лету, что упрощает модель данных, логику обработки и конкурентный доступ

Колоночное хранение против строчного

Выбор способа хранения данных зависит от ситуации …

Использовать column store:

• Обычно в вычислениях участвует одно поле или несколько полей

• Поиск по таблице выполняется по нескольким полям

• Таблица содержит большое количество полей

• Таблица содержит большое количество записей и требуется выполнение колоночных операций (агрегация, скан, т.д.)

• Возможен высокий уровень компрессии из-за низкой кардинальности данных (в сравнении с количеством записей)

Использовать row store:

• Приложение одномоментно обрабатывает одну или несколько строк таблицы (много select'ов и/или update'ов по единичным записям)

• Приложению обычно требуется доступ к отдельной записи или строке

• Поля таблицы содержат данные высокой кардинальности (низкая компрессия)

• Не требуется агрегация или поиск данных

• Таблица содержит небольшое количество записей (конфигурационные таблицы).