BIGDATA

Лекция 8

ПЛАН ЛЕКЦИИ

- ClickHouse

ВЗГЛЯД ИЗВНЕ

- Все очень похоже на классический SQL
- Таблицы, запросы
- И вроде меняются данные без явного торможения
- И маленькие запросы тоже сильно не тормозят

ПОВНИМАТЕЛЬНЕЕ

- Внимательным взглядом можно ухватить отличия
- Другой набор типов данных
- Разнообразие способов загрузки данных
- Но это не выглядит как что-то принципиальное

ПРИМЕР

```
CREATE TABLE hits_UserID_URL
(
    `UserID` UInt32,
    `URL` String,
    `EventTime` DateTime
)
ENGINE = MergeTree
PRIMARY KEY (UserID, URL)
ORDER BY (UserID, URL, EventTime)
SETTINGS index_granularity = 8192,
index_granularity_bytes = 0, compress_primary_key = 0;
```

ПОВНИМАТЕЛЬНЕЕ

- Можем видеть указание ENGINE
- Но в SQL тоже есть различение абстракции таблицы и способа хранения
- В Postgres реже оно указывается - потому что есть типовое умолчание
- А в MySQL - чаще (InnoDB/MyISAM)

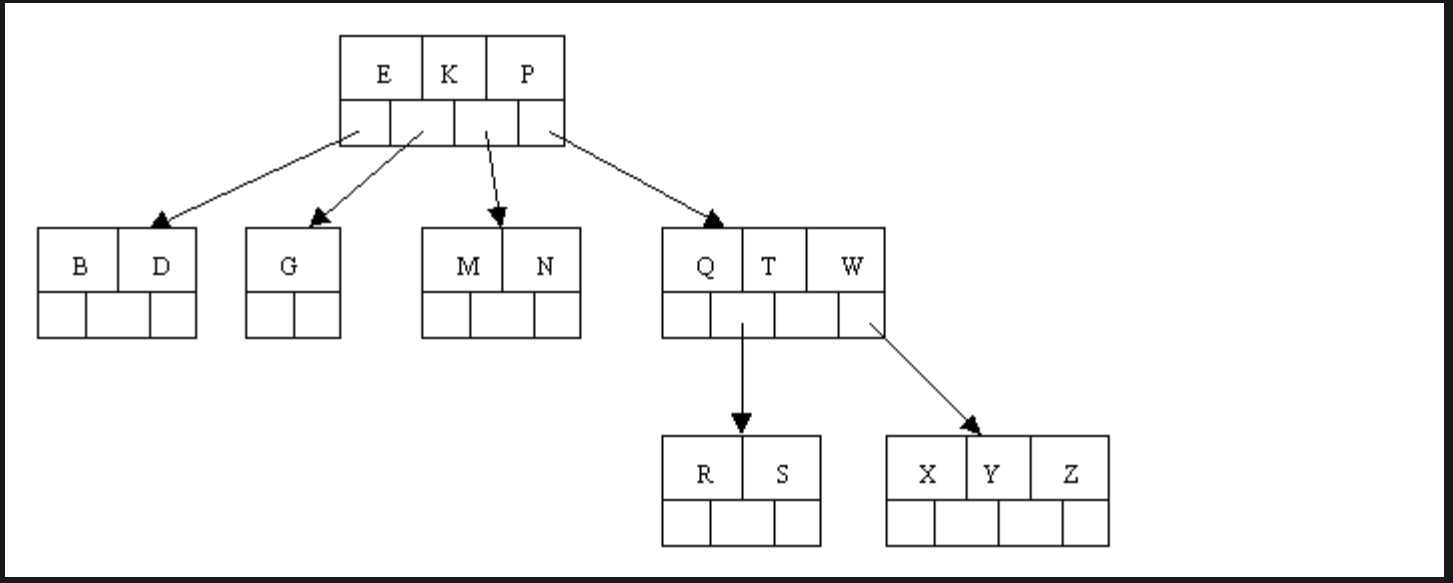
PRIMARY KEY

- Не похоже на Primary Key в SQL-понимании
- В условном Postgres-е PRIMARY KEY имеет двойное назначение
- Это инструмент нормализации и одновременно способ задания индекса
- В ClickHouse это только про индексирование

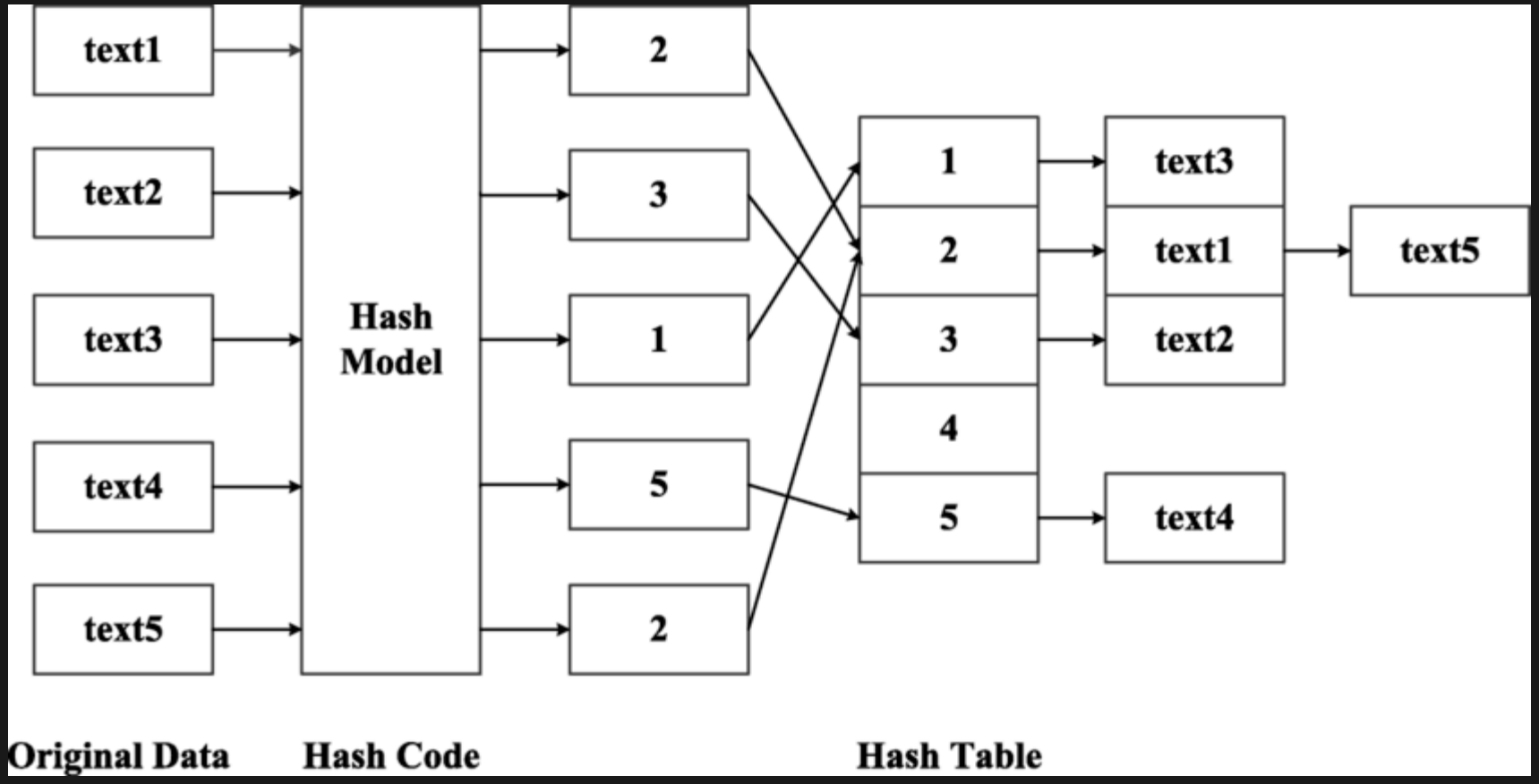
ИНДЕКСИРОВАНИЕ

- Но и подходы к индексированию отличаются
- Postgres: через B-дерево или хеш-индекс
- Которые хорошо работают для точечного поиска
- B-дерево умеет и в диапазонный поиск, но есть нюансы

B-ДЕРЕВЬЯ



ХЕШ-ИНДЕКСЫ


text1
text2
text3
text4
text5
Hash Model
2
3
1
5
2
1
2
3
4
5
text3
text1
text2
text4
text5
Original Data
Hash Code
Hash Table

БЕЗ UNIQUE

- Неуникальность невысокая - при точном поиске по ключу она хорошо работают
- Чем больше значений налипает на ключ - тем глубже становится дерево
- Для B-дерева налипание можно устранить удлинением ключа
- Но еще надо придумать - как именно
- И это углубляет дерево

БЕЗ UNIQUE

- А за записями еще надо бегать
- Выборка в 1/20 от объема базы может потребовать обхода всей базы
- Если запись - 1/20 от страницы
- И записи выборки распределены равномерно

ЧТО ПРЕДЛАГАЕТ CH

- Организацию данных, приспособленную под ситуацию "много записей на одном ключе"
- Более широко - возможность создавать разные модификации организации данных
- MergeTree - базовая модель
- Разберемся в ней

ЧТО ПРЕДЛАГАЕТ CH

- Понятия: парты (не партиции, хотя и они есть) и гранулы
- За партами можно наблюдать через системную таблицу system.parts
- На свежесозданном сервере она пуста
- И даже факт создания таблиц ее не наполняет

ДОБАВЛЕНИЕ ДАННЫХ

- Каждый INSERT создает парт
- Каждый парт занимает свой каталог
- Худшее, что можно придумать в такой модели - это вставлять данные по одной записи
- Парты потом могут сливаться - но не сразу

ДОБАВЛЕНИЕ ДАННЫХ

- Внутри парта - колоночная организация
- И упорядочивание по первичному ключу
- В целом большие парты - это хорошо
- Но в них еше нужно быстро находить нужное

ДОБАВЛЕНИЕ ДАННЫХ

- В классических техниках индексирования нужно заводить по B-дереву на парт
- Но это издержки - неоправданные
- У нас же сплошная область - и надо этим воспользоваться
- А если вытаскиваем большими кусками - не так важно точное позиционирование начала и конца
- Важнее компактность индекса

ДОБАВЛЕНИЕ ДАННЫХ

- Заведем разреженный бинарный поиск
- Поделим набор записей парта на куски по количеству записей
- В индексе храним значение ключа в началах гранул
- И начало гранулы для каждой колонки

ИЗВЛЕЧЕНИЕ ДАННЫХ

- Идем в нужные парты
- Определяем гранулы-кандидаты
- Вычитываем гранулы нужных колонок
- Из краевых гранул выбираем нужный диапазон
- Фильтруем

КОГДА БУДЕТ ПЛОХО

- Когда размер выдачи запроса заметно меньше гранулы
- По умолчанию гранула - 8K записей
- Или из-за мелкости гранул индексы партов увеличивается
- Но все равно выигрыш возможен за счет вертикального хранения

ДРУГИЕ СЦЕНАРИИ

- Ищем по первичному ключу - есть шансы на успех
- По префиксу первичного ключа - тоже
- А если по по второму полю составного ?
- Есть шанс, если первое - "low cardinality"

ТИПЫ ДАННЫХ

- В целом в рамках MergeTree-модели крайне важен выбор первичного ключа
- Разберем особенности типов данных, важные с этой точки зрения
- И с точки зрения компактности хранения

ТИПЫ ДАННЫХ

- В SQL NULL возможен по умолчанию, надо специально запрещать
- В CH это атрибут типа, по умолчанию отсутствующий
- На nullable тип расходуется отдельная скрытая колонка

ТИПЫ ДАННЫХ

- Для ограниченного набора значений лучше использовать enum
- Это поможет более плотной упаковке колонок
- Если набор значений трудно определить заранее - может помочь атрибут типа LowCardinality
- LowCardinality-строка хранится как номер слова в словаре

ТИПЫ ДАННЫХ

- До 10 000 значений LowCardinality выглядит как удачный вариант
- Дальше издержки начинают превышать выгоду
- Строка в primary key - спорная идея
- Особенно первым полем в составном

ТИПЫ ДАННЫХ

- Структурированная строка (URL, иерархический путь) вторым-третьим полем - обсуждаемо
- Low-Cardinality-строка (теги и т.п.) - даже первым номером может подойти
- Тип сообщения, номер паспорта - скорее нет (каждая по своим причинам)
- Даты, таймстемпы - хорошие кандидаты
- Но рассмотреть вариант понижения кардинальности

ДОБАВЛЕНИЕ ДАННЫХ

- Первая развилка - table functon vs engine
- Engine - реализация интерфейса к формату хранения
- Можно к внутреннему (MergeTree), а можно - к внешнему (Kafka)
- Table function - интерфейс к внешнему источнику для перебрасывания данных во внутренний формат

ДОБАВЛЕНИЕ ДАННЫХ

- Выбор в пользу table function порождает другую независимую развилку
- Из какого формата данные читаем: csv, avro и т.п.
- Есть тонна вариантов по каждой из осей
- Можно одним запросом загрузить архивированный csv-файл из S3 хранилища
- Он еще схему породит с выведением типов по содержимому

ПАРТИЦИОНИРОВАНИЕ

- Еще один уровень разбиения
- При добавлении данные разбиваются по партициям
- А внутри партиций - по партам
- Слияние партов происходит внутри партиции

ПАРТИЦИОНИРОВАНИЕ

- Партициями легко удалять
- Партами тоже легко, но их содержимое более динамично
- Партиции ускорять поиск - если запрос подразумевает поиск в узком наборе партиций
- Могут замедлять - если запросы размазываются по многим партициям (+ блокирование слияния партов)

РАСПРЕДЕЛЕННЫЕ ТАБЛИЦЫ

- Специальный engine - Distributed
- Указываем ключ шардирования
- Должен способствовать распределению нагрузки
- Но не мельчить парты

РАСПРЕДЕЛЕННЫЕ ТАБЛИЦЫ

- Внутри шардов могут быть реплики
- Много разных вариантов настроек
- Пример: distributed_group_by_no_merge при группировке
- Не мерджим результаты между шардами

KAK TAM C ACID

КАК ТАМ С ACID

- Если много партиций - гарантии только внутри партиции
- Если много шардов - гарантии только внутри шарда
- Нет полновесного BEGIN/ROLLBACK/COMMIT
- Есть экспериментальный и ограниченный

ВАРИАЦИИ MERGETREE

- SummingMergeTree - хранит просуммированные значения по первичному ключу
- Для выбранных колонок - модель хранения позволяет делать для каждой колонки свой выбор
- Полезно в сочетании с сохранением большого числа записей в запросе
- Несуммируемые поля не должны доминировать в объеме

ПРИМЕР

```
1 CREATE TABLE summtt
2 (
3     key UInt32,
4     value UInt32
5 )
6 ENGINE = SummingMergeTree()
7 ORDER BY key
```

ПРИМЕР

```
 1 CREATE TABLE nested_sum
 2 (
 3     date Date,
 4     site UInt32,
 5     hitsMap Nested(
 6         browser LowCardinality(String),
 7         imps UInt32,
 8         clicks UInt32
 9     )
10 ) ENGINE = SummingMergeTree
11 PRIMARY KEY (date, site);
```

РАЗВИТИЕ И ОБОБЩЕНИЕ

- Агрегация - это не только суммирование
- Агрегацию хотелось бы предвычислять
- Исходные значения терять не хотелось бы
- Но хранить отдельно - чтобы не убивался эффект от предвычисления

ПРИМЕР

```
1 CREATE DATABASE test;
2
3 CREATE TABLE test.visits
4  (
5     StartDate DateTime,
6     CounterID UInt64,
7     Sign Int32,
8     UserID Int32
9 ) ENGINE = MergeTree ORDER BY (StartDate, CounterID);
```

ПРИМЕР

```
CREATE TABLE test.agg_visits (
    StartDate DateTime,
    CounterID UInt64,
    Visits AggregateFunction(sum, Int32),
    Users AggregateFunction(uniq, Int32)
)
ENGINE = AggregatingMergeTree()
ORDER BY (StartDate, CounterID);
```

ПРИМЕР

```
1 CREATE MATERIALIZED VIEW test.visits_mv TO test.agg_visits
2 AS SELECT
3     StartDate,
4     CounterID,
5     sumState(Sign) AS Visits,
6     uniqState(UserID) AS Users
7 FROM test.visits
8 GROUP BY StartDate, CounterID;
```

ПРИМЕР

```sql
INSERT INTO test.visits
(StartDate, CounterID, Sign, UserID)
 VALUES (1667446031000, 1, 3, 4),
        (1667446031000, 1, 6, 3);

SELECT
    StartDate,
    sumMerge(Visits) AS Visits,
    uniqMerge(Users) AS Users
FROM test.agg_visits
GROUP BY StartDate
ORDER BY StartDate;
```

MATERIALIZED VIEW

- Ключевой инструмент ClickHouse
- Инструмент дуплицирования данных для ускорения запросов
- Не транзакционен в целом
- Транзакциононен внутри таблицы/партиции/шарда

ОБНОВЛЕНИЯ

- Найти обновляемые записи не сложнее, чем в SELECT
- Сложно обновить - хотя зависит от числа обновляемых полей
- Предлагается три варианта
- Update mutations, Lightweight updates, ReplacingMergeTree

ОБНОВЛЕНИЯ

- Но это если меняются не поля первичного ключа
- Их менять можно, но сложнее
- Есть еще один, нулевой вариант: обойтись без изменений
- Подварианты: корректирующие вставки, коррекция на стороне клиента

UPDATE MUTATIONS

```
1 ALTER TABLE posts_temp
2     (UPDATE AnswerCount = AnswerCount + 1
3      WHERE AnswerCount = 0)
```

- Запускается асинхронно
- Мониторится через system.mutations

UPDATE MUTATIONS

- Идет по партам и все перелопачивает
- Для wide-формата хранения скорее будет быстрее
- Не изолированно от параллельно работающих SELECT-ов
- Атомарно на уровне партов

LIGHTWEIGHT UPDATES

- Только в ClickHouse cloud
- Включается параметром запроса - `apply_mutations_on_fly `
- Так себе "lightweight" по сути - но имеют существенное преимущество
- Изменения хранятся в виде "заплатки"
- Знание о том, какие записи и как меняются

LIGHTWEIGHT UPDATES

- Изменения вступают в действие атомарно
- При исполнении SELECT-ов вносятся поправки
- Одновременно фоново эти правки применяются
- Примененные части заплаток выбрасываются

SPARK STREAMS

- Данные не в стационарном источнике
- Данные постепенно приходят
- Хочется обрабатывать их в стиле привычных операций

ПРИМЕР

```
1 from pyspark.sql import SparkSession
2 from pyspark.sql.functions import explode
3 from pyspark.sql.functions import split
4
5 spark = SparkSession \
6     .builder \
7     .appName("StructuredNetworkWordCount") \
8     .getOrCreate()
```

ПРИМЕР

```
# в другом терминале запустим: nc -lk 9999

lines = spark \
    .readStream \
    .format("socket") \
    .option("host", "localhost") \
    .option("port", 9999) \
    .load()
```

ПРИМЕР

```
# Split the lines into words
words = lines.select(
   explode(
       split(lines.value, " ")
   ).alias("word")
)

# Generate running word count
wordCounts = words.groupBy("word").count()
```

ПРИМЕР

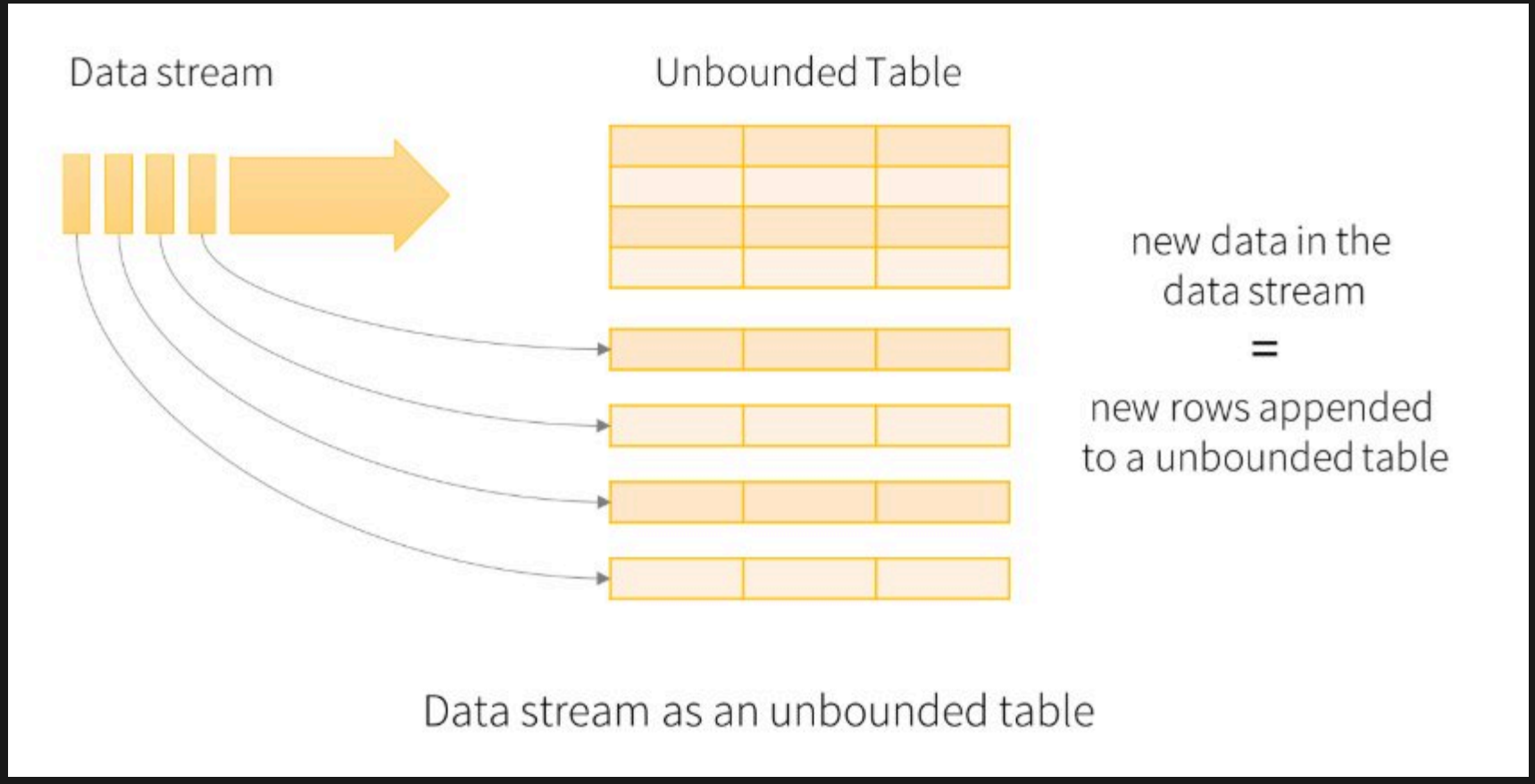
```
query = wordCounts \
    .writeStream \
    .outputMode("complete") \
    .format("console") \
    .start()

query.awaitTermination()
```

МОДЕЛЬ ВЫЧИСЛЕНИЙ

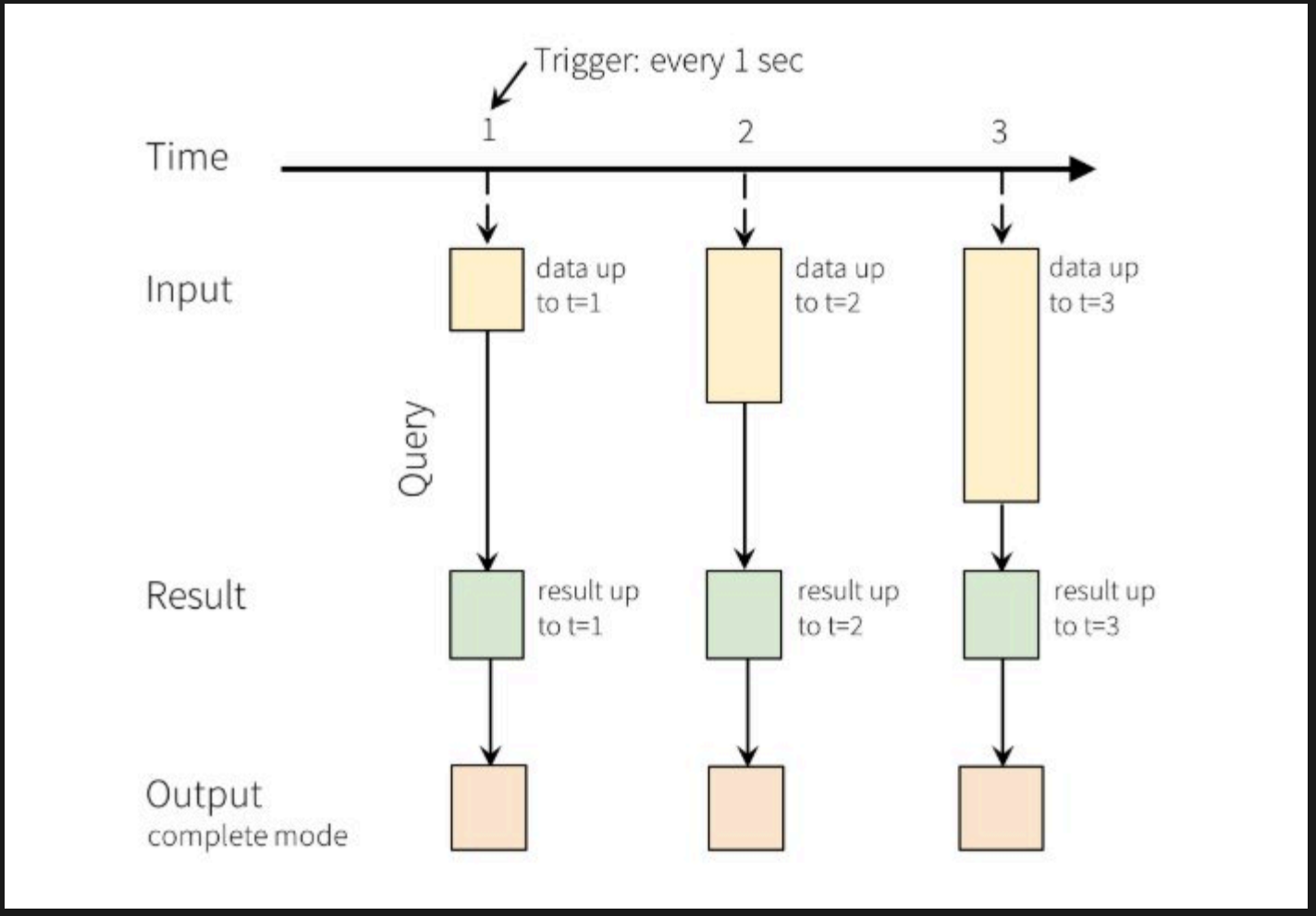
- В пакетной обработке (RDD, DataFrame) запрос сначала конструируется, потом исполняется
- Здесь - динамически пополняющаяся подкапотная табличка
- Дополнения собираются
- Запрос инкрементально пересчитывается

КАРТИНКА



Data stream as an unbounded table

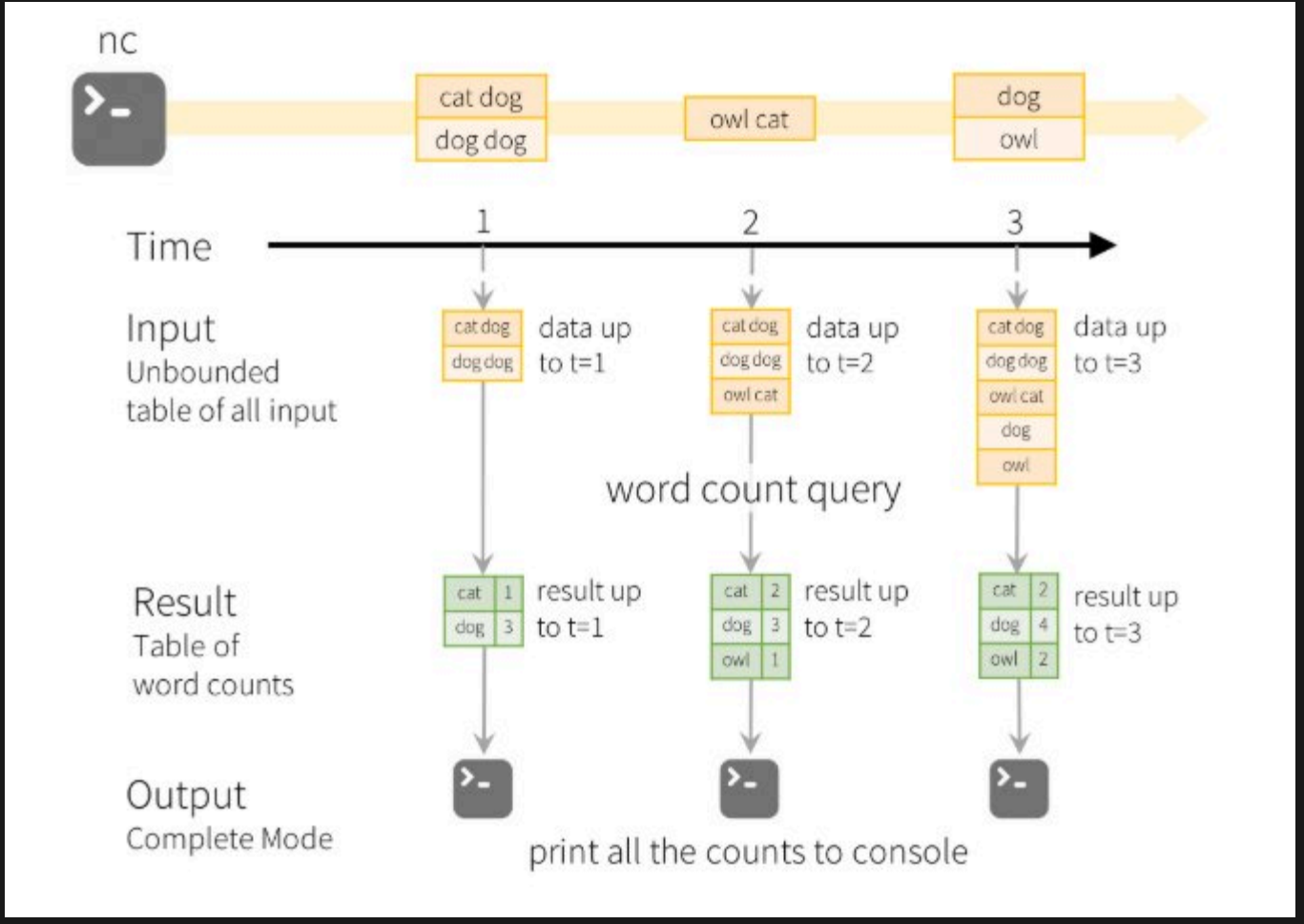
КАРТИНКА


Trigger: every 1 sec
Time
1
2
3
Input
data up to t=1
data up to t=2
data up to t=3
Query
Result
result up to t=1
result up to t=2
result up to t=3
Output
complete mode

МОДЕЛЬ ВЫЧИСЛЕНИЙ

- Три режима вывода
  - Complete
  - Append
  - Update

КАРТИНКА


nc
cat dog
dog dog
owl cat
dog
owl
Time
1
2
3
Input
Unbounded
table of all input
cat dog
dog dog
data up
to t=1
cat dog
dog dog
owl cat
data up
to t=2
cat dog
dog dog
owl cat
dog
owl
data up
to t=3
word count query
Result
Table of
word counts
cat 1
dog 3
result up
to t=1
cat 2
dog 3
owl 1
result up
to t=2
cat 2
dog 4
owl 2
result up
to t=3
Output
Complete Mode
print all the counts to console

МОДЕЛЬ ВЫЧИСЛЕНИЙ

- Вся таблица реально не хранится
- Хранится результат
- И то, что нужно для его обновления