

## Содержание

<b>1 Экзамен</b>	<b>3</b>
1.1 Базы данных и системы управления базами данных. Определения, основные функции и классификация . . . . .	3
1.1.1 Основные функции СУБД . . . . .	4
1.1.2 Классификация СУБД . . . . .	4
1.1.3 Архитектура хранения данных . . . . .	5
1.2 Семантическое моделирование данных . . . . .	6
1.3 Реляционная модель данных: структурная, целостная, манипуляционная части. Реляционная алгебра. Исчисление кортежей . . . . .	7
1.4 Теория проектирования реляционных баз данных: функциональные зависимости, нормальные формы . . . . .	10
1.4.1 Функциональная зависимость . . . . .	11
1.4.2 Дополнительные правила вывода. . . . .	11
1.4.3 Общая схема процедуры нормализации . . . . .	12
1.5 Теория проектирования хранилищ данных. Основные принципы построения. ETL и ELT процессы . . . . .	13
1.6 Транзакции. Определение, свойства и уровни изоляции транзакций. Неблагоприятные эффекты, вызванные параллельным выполнением транзакции, и способы их устранения. Управление транзакциями и способы обработки ошибок . . . . .	16
1.7 Блокировки. Определение, свойства, иерархии, гранулярность и взаимоблокировки, алгоритмы обнаружения взаимоблокировок . . . . .	21
1.7.1 Иерархии . . . . .	21
1.7.2 Гранулярность . . . . .	21
1.7.3 Взаимоблокировки . . . . .	22
1.8 Журнализация. Операции журнала транзакций и его логическая и физическая архитектуры. Модели восстановления. Метаданные . . . . .	24
1.8.1 Типы используемых файлов . . . . .	24
1.8.2 Операции . . . . .	24
1.8.3 Логическая архитектура . . . . .	25
1.8.4 Физическая архитектура . . . . .	25
1.8.5 Модель восстановления . . . . .	26
1.9 Безопасность и Аудит. Ключевые понятия и участники системы безопасности. Модели управления доступом . . . . .	29

1.9.1	Участники системы безопасности . . . . .	29
1.9.2	Защищаемые объекты . . . . .	30
1.9.3	Разрешения . . . . .	30
1.9.4	Аудит SQL Server . . . . .	31
1.9.5	Система управления на основе политик . . . . .	31
1.9.6	Режим оценки политик . . . . .	31
1.9.7	Преимущества системы управления на основе политик . . . . .	32
1.10	MPP системы. Распределенное и колоночное хранение. Распределенные вычисления, модель MapReduce. Обеспечение отказоустойчивости. . . . .	33
1.10.1	Массивно-параллельные системы обработки (massively parallel processing, MPP). . . . .	33
1.10.2	Аппаратная архитектура на примере VERTICA . . . . .	34
1.10.3	Проекции . . . . .	35
1.10.4	Отказоустойчивость . . . . .	36
1.10.5	В чем выгода колоночного хранения? . . . . .	37
1.11	In-Memory базы данных. Преимущества и недостатки. Примеры использования . . . . .	39
1.12	Инструкции языка описания данных, инструкции языка обработки данных, инструкции безопасности, инструкции управления транзакциями . . . . .	40
1.13	Объекты базы данных: функции, процедуры, триггеры и курсоры . . . . .	44
1.13.1	Функции . . . . .	44
1.13.2	Хранимая процедура . . . . .	46
1.13.3	Триггеры . . . . .	48
1.13.4	Курсоры . . . . .	49
1.14	Оптимизация запроса: индексы, партиционирование, сегментирование . . . . .	53
1.14.1	Индексы . . . . .	53
1.14.2	Партиционирование . . . . .	54
1.14.3	Сегментирование . . . . .	55
1.15	План запроса. Этапы выполнения запроса . . . . .	57
1.15.1	Пять стадий выполнения запроса . . . . .	58
1.15.2	План запроса . . . . .	59
1.15.3	Как вызвать план запроса в postgres: . . . . .	59

# 1 Экзамен

## 1.1 Базы данных и системы управления базами данных. Определения, основные функции и классификация

**База данных (БД)** — самодокументированное собрание интегрированных записей.

- 1) БД является самодокументированной, т.е. она содержит описание собственной структуры. Это описание называется словарем данных, каталогом данных или метаданными.
- 2) БД является собранием интегрированных записей, т.е. она содержит:
  - файлы данных;
  - метаданные (данные о данных);
  - индексы, которые представляют связи между данными, а также служат для повышения производительности приложений базы данных;
  - метаданные приложения.
- 3) БД является информационной моделью пользовательской модели предметной области.

**OLAP** — online analytic processing (операционные данные).

**OLTP** — online transaction processing (перманентные данные).

OLAP	OLTP
чтение	вставка, удаление, обновление
минимальное время отклика	минимальное время отклика на эти операции

**Транзакции** — либо все действия, либо никакие действия.

**Любая бд хранит:**

- 1) метаданные (данные о данных);
- 2) файлы данных,
- 3) Индексы (indexes), которые представляют связи между данными, а также служат для повышения производительности приложений базы данных.
- 4) Может содержать метаданные приложений (application metadata).

**Основные характеристики, требования**

- 1) **Неизбыточность данных** — каждое данное присутствует в БД в единственном экземпляре.
- 2) **Совместное использование данных** многими пользователями.
- 3) **Эффективность доступа к БД** - высокое быстродействие, т. е. малое время отклика на запрос.

- 4) **Целостность данных** — соответствие имеющейся в БД информации её внутренней логике, структуре и всем явно заданным правилам.
- 5) **Безопасность данных** — защита данных от преднамеренного или непреднамеренного искажения или разрушения данных.
- 6) **Восстановление данных** после программных и аппаратных сбоев.
- 7) **Независимость данных** от прикладных программ.

**Система баз данных (СБД)** — совокупность одной или нескольких баз данных и комплекса информационных, программных и технических средств, обеспечивающих накопление, обновление, корректировку и многоаспектное использование данных в интересах пользователей.

*Система управления базами данных (СУБД)* — приложение, обеспечивающее создание, хранение, обновление и поиск информации в базах данных.

#### **1.1.1 Основные функции СУБД**

- 1) Управление данными во внешней памяти.
- 2) Управление буферами оперативной памяти.
- 3) Управление транзакциями.
- 4) Журнализация.
- 5) Поддержка языка или языкового пакета (-ов).

#### **1.1.2 Классификация СУБД**

- 1) По модели данных:
  - Дореляционные (Инвертированные списки, иерархические и сетевые)
    - Инвертированные списки (файлы). БД на основе инвертированных списков представляет собой совокупность файлов, содержащих записи (таблиц). Для записей в файле определен некоторый порядок, диктуемый физической организацией данных. Для каждого файла может быть определено произвольное число других упорядочений на основании значений некоторых полей записей (инвертированных списков). Обычно для этого используются индексы. В такой модели данных отсутствуют ограничения целостности как таковые. Все ограничения на возможные экземпляры БД задаются теми программами, которые работают с БД. Одно из немногих ограничений, которое все-таки может присутствовать - это ограничение, задаваемое уникальным индексом.
    - Иерархические
    - Сетевые (могут быть представлены в виде графа; логика выборки зависит от физической организации данных)

- Реляционные
  - Структурный (данные — набор отношений)
  - Целостностный (отношения (таблицы) отвечают определенным условиям целостности)
  - Манипуляционный (манипулирование отношениями осуществляется средствами реляционной алгебры и/или реляционного исчисления)
- Постреляционные

2) По архитектуре организации хранения данных:

- Локальные (все части локальной СУБД размещаются на одном компьютере)
- Распределенные (части СУБД могут размещаться на 2-х и более компьютерах)

3) По способу доступа к БД:

- Файл-серверные (при работе с базой, данные перегоняются приложению, которое с ней работает, вне зависимости от того, сколько их нужно. Все операции — на стороне клиента. Файловый сервер периодически обновляется тем же клиентом)
- Клиент-серверные (вся работа на сервере, по сети передаются результаты запросов, гораздо меньше информации. Обеспечивается безопасность данных, потому что все происходит на стороне сервера. Проще исключить одновременное изменение и тп)
- Встраиваемые — библиотека, которая позволяет унифицированным образом хранить большие объемы данных на локальной машине. Доступ к данным может происходить через SQL либо через особые функции СУБД. Встраиваемые СУБД быстрее обычных клиент-серверных и не требуют установки сервера, поэтому востребованы в локальном ПО, которое имеет дело с большими объемами данных.
- Сервисно-ориентированные (БД является хранилищем сообщений, промежуточных состояний, метаданных об очередях сообщений и сервисах)
- Прочие (пространственная, временная и пространственно-временная)

### 1.1.3 Архитектура хранения данных

1) Локальные.

2) Распределенные.

3) По способу обращения к данным.

- Файл серверные.
- Клиент серверные (PostGress, MSSQL, Oracle, MySQL, Mongo).
- Встраиваемые (SQLite).
- Сервисно-ориентированные (KafkaBD).
- Прочее - time series.

## 1.2 Семантическое моделирование данных

Любая развитая семантическая модель данных, как и реляционная модель, включает структурную, манипуляционную и целостную части. Главным назначением семантических моделей является обеспечение возможности выражения семантики данных. На практике семантическое моделирование используется на первой стадии проектирования базы данных. При этом в терминах семантической модели производится концептуальная схема базы данных, которая затем:

- 1) Либо вручную преобразуется к реляционной схеме;
- 2) Либо реализуется автоматизированная компиляция концептуальной схемы в реляц.;
- 3) Либо происходит работа с базой данных в семантической модели, т.е. под управлением СУБД, основанных на семантических моделях данных.

Наиболее известным представителем класса семантических моделей предметной области является модель «сущность-связь» или ER-модель, предложенная Питером Ченом в 1976 году.

Модель сущность-связь — модель данных, позволяющая описывать концептуальные схемы предметной области. Предметная область — часть реального мира, рассматриваемая в пределах данного контекста. Под контекстом здесь может пониматься, например, область исследования или область, которая является объектом некоторой деятельности. ER-модель используется при высокоуровневом (концептуальном) проектировании баз данных.

Основными понятиями ER-модели являются сущность, связь и атрибут(свойство).

Сущность — это реальный или представляемый объект, информация о котором должна сохраняться и быть доступна. При этом имя сущности - имя типа, а не некоторого конкретного экземпляра этого типа. Каждый экземпляр сущности должен быть отличим от любого другого экземпляра этой сущности.

Связь — это ассоциация, устанавливаемая между сущностями. Эта ассоциация может существовать между разными сущностями или между сущностью и ей же самой (рекурсивная связь). Сущности, включенные в связь, называются её участниками, а количество участников - степенью связи. Связи в ER-модели могут иметь тип «один к одному», «один ко многим», «многие ко многим».

Свойством/атрибутом сущности (и связи) является любая деталь, которая служит для уточнения, идентификации, классификации, числовой характеристики или выражения состояния сущности (или связи).

Ключи: Первичный ключ — набор атрибутов однозначно идентифицирующий кортеж значений, и Внешний ключ

### 1.3 Реляционная модель данных: структурная, целостная, манипуляционная части. Реляционная алгебра. Исчисление кортежей

Реляционная модель данных согласно трактовке Кристофера Дейта состоит из трех частей: структурной, целостной и манипуляционной.

**Структурная часть** описывает из каких объектов состоит реляционная модель. Основной структурой данных в реляционной модели является нормализованные  $n$ -мерные отношения и основными понятиями структурной части реляционной модели является:

- *Тип данных* — множество значений и операций над ними. (понятие такое же как и в языках программирования);
- *Домен* можно считать уточнением типа данных и рассматривать как подмножество значений некоторого типа данных, имеющий определенный смысл. Характеризуется следующими свойствами:
  - имеет уникальное имя в пределах базы данных;
  - определен на некотором типе данных или на другом домене;
  - может иметь логическое условие, позволяющее описать подмножество данных, доступных для данного домена;
  - несет определенную смысловую нагрузку

Домен отражает семантику, определенной предметной области и может быть не сколько доменов совпадающих как подмножество, но с различным смыслом. Основное значения домена ограничивается сравнением.

- *Атрибут отношения* — это пара вида  $\langle \text{имя\_атрибута}, \text{имя\_домена} \rangle$ , при этом имена атрибутов должны быть уникальны в пределах отношения, но могут совпадать с именем домена.
- *Схема отношения* — это именованное множество упорядоченных пар  $\langle \text{имя\_атрибута}, \text{имя\_домена} \rangle$ .
- *Схема БД* — это множество именованных схем отношений.
- *Кортеж* — это множество упорядоченных пар  $\langle \text{имя\_атрибута}, \text{значение\_атрибута} \rangle$ , которое содержит одно вхождение каждого имени атрибута, принадлежащего схеме отношения.
- *Отношение* определенное на множестве из  $n$  доменов (не обязательно различных), содержит две части: заголовок (схему отношения) и тело (множество из  $m$  кортежей). Значения  $n$  и  $m$  называются соответственно степенью и кардинальностью отношения.
- Непустое подмножество множества атрибутов схемы отношения будет *потенциальным*

*ключом*  $\Leftrightarrow$  оно будет обладать свойствами уникальности (в отношении нет двух различных кортежей с одинаковыми значениями потенциального ключа) и избыточности (никакое из собственных подмножеств множества потенциального ключа не обладает свойством уникальности).

- В реляционной модели по традиции один из потенциальных ключей должен быть выбран в качестве *первичного ключа*, а все остальные потенциальные ключи будут называться *альтернативными*.
- *Реляционная база данных* — это набор отношений, имена которых совпадают с именами схем отношений в схеме базы данных.

**Целостностная часть** описывает ограничения специального вида, которые должны выполняться для любых отношений в любых реляционных базах данных. Это целостность сущностей и целостность внешних ключей.

**Манипуляционная часть** описывает два эквивалентных способа манипулирования реляционными данными - реляционную алгебру и реляционное исчисление.

**Реляционная алгебра** является основным компонентом реляционной модели, опубликованной Коддом, и состоит из восьми операторов, составляющих две группы по четыре оператора:

- Традиционные операции над множествами: объединение (UNION), пересечение (INTERSECT), разность (MINUS) и декартово произведение (TIMES). Все операции модифицированы, с учетом того, что их операндами являются отношения, а не произвольные множества.
- Специальные реляционные операции: ограничение (WHERE), проекция (PROJECT), соединение (JOIN) и деление (DIVIDE BY).

Результат выполнения любой операции реляционной алгебры над отношениями также является отношением. Эта особенность называется свойством реляционной замкнутости. Утверждается, что поскольку реляционная алгебра является замкнутой, то в реляционных выражениях можно использовать вложенные выражения сколь угодно сложной структуры.

**Исчисление** существует в двух формах: исчисление кортежей и исчисление доменов. Основное различие между ними состоит в том, что переменные исчисления кортежей являются переменными кортежей (они изменяются на отношении, а их значения являются кортежами), в то время как переменные исчисления доменов являются переменными доменов (они изменяются на доменах, а их значения являются скалярами).

Выражение исчисления кортежей содержит заключенный в скобки список целевых элементов и выражение WHERE, содержащее формулу WFF ("правильно построенную формулу"). Такая формула WFF составляется из кванторов (EXISTS и FORALL), свободных и свя-



занных переменных, литералов, операторов сравнения, логических (булевых) операторов и скобок. Каждая свободная переменная, которая встречается в формуле WFF, должна быть также перечислена в списке целевых элементов.

## 1.4 Теория проектирования реляционных баз данных: функциональные зависимости, нормальные формы

### Теория проектирования реляционных баз данных

При проектировании баз данных решаются две основные проблемы:

- проблема логического проектирования баз данных;
- проблема физического проектирования баз данных.

Классический подход к проектированию реляционных баз данных заключается в том, что сначала предметная область представляется в виде одного или нескольких отношений, а далее осуществляется процесс *нормализации* схем отношений, причем каждая следующая нормальная форма обладает лучшими свойствами, чем предыдущая. Каждой нормальной форме соответствует некоторый определенный набор ограничений, и отношение находится в некоторой нормальной форме, если удовлетворяет свойственному ей набору ограничений. В теории реляционных баз данных обычно выделяется следующая последовательность нормальных форм:

- 1) *Первая нормальная форма (1НФ/1NF)*. Отношение находится в 1НФ, если все его атрибуты являются простыми, все используемые домены должны содержать только скалярные значения, при этом нет повторяющихся кортежей.
- 2) *Вторая нормальная форма (2НФ/2NF)*. Отношение находится во 2НФ, если оно находится в 1НФ и каждый не ключевой атрибут неприводимо зависит от первичного ключа, т.е. в составе потенциального ключа отсутствует подмножество атрибутов, от которых также можно вывести неключевые атрибуты.
- 3) *Третья нормальная форма (3НФ/3NF)*. Отношение находится в 3НФ, когда находится во 2НФ и каждый нетривиально зависит от первичного ключа.
- 4) *Нормальная форма Бойса-Кодда (НФБК)*. Отношение находится в НФБК, когда каждая нетривиальная и неприводимая слева функциональная зависимость обладает потенциальным ключом в качестве детерминанта.
- 5) *Четвертая нормальная форма (4НФ/4NF)*. Отношение находится в НФБК, если оно находится в НФБК и все нетривиальные многозначные зависимости фактически являются функциональными зависимостями от ее потенциальных ключей.
- 6) *Пятая нормальная форма (5НФ/5NF)*. Отношение находится в 5НФ  $\Leftrightarrow$  каждая нетривиальная зависимость соединения определяется потенциальным ключом этого отношения.

**Основные свойства** нормальных форм:

- 1) каждая следующая НФ в некотором смысле лучше предыдущей;
- 2) при переходе к следующей НФ свойства предыдущих нормальных свойств сохраняются.

Процесс проектирования реляционной базы данных на основе метода нормализации преследует две основные цели: избежать избыточность хранения данных и устранить аномалии обновления отношений (под этим подразумеваются определенные трудности, появляющиеся при выполнении операций обновления INSERT, DELETE, UPDATE).

Наиболее важные на практике нормальные формы отношений основываются на фундаментальном в теории реляционных баз данных понятии функциональной зависимости.

#### 1.4.1 Функциональная зависимость

Пусть  $R$  - это отношение, а  $X$  и  $Y$  - произвольные подмножества множества атрибутов отношения  $R$ . Тогда  $Y$  функционально зависимо (ФЗ) от  $X$ , что в символическом виде записывается как  $X \rightarrow Y \Leftrightarrow \forall \text{значение множества } X \text{ связано в точности с одним значением множества } Y$ . (Левая ФЗ часть — детерминант, правая — зависимая часть)

Очевидным способом сокращения размера множества ФЗ было бы исключение тривиальных зависимостей, т. е. таких, которые не могут не выполняться.

Множество всех ФЗ, которые задаются данным множеством ФЗ  $S$ , называется замыканием  $S$  и обозначается символом  $S^+$ .

Пусть в перечисленных ниже правилах  $A$ ,  $B$  и  $C$  — произвольные подмножества множества атрибутов заданной переменной-отношения  $R$ , а символическая запись  $AB$  означает  $\{A, B\}$ . Тогда правила вывода определяются следующим образом:

**Теорема** (Правила вывода).

- 1) *Правило рефлексивности*  $(B \subseteq A) \Rightarrow (A \rightarrow B)$
- 2) *Правило дополнения*  $(A \rightarrow B) \Rightarrow (AC \rightarrow BC)$
- 3) *Правило транзитивности*  $(A \rightarrow B) \&\& (B \rightarrow C) \Rightarrow (A \rightarrow C)$

Каждое из этих правил может быть непосредственно доказано на основе определения ФЗ. Более того, эти правила являются полными в том смысле, что для заданного множества ФЗ  $S$  минимальный набор ФЗ, которые подразумевают все зависимости из множества  $S$ , может быть выведен из  $S$  на основе этих правил. Они также являются исчерпывающими, поскольку никакие дополнительные ФЗ (т.е. ФЗ, которые не подразумеваются ФЗ множества  $S$ ) с их помощью не могут быть выведены. Иначе говоря, эти правила могут быть использованы для получения замыкания  $S^+$ .

#### 1.4.2 Дополнительные правила вывода.

- 1) *Правило самоопределения*  $(A \rightarrow A)$
- 2) *Правило декомпозиции*  $(A \rightarrow B) \Rightarrow (A \rightarrow B) \&\& (A \rightarrow C)$
- 3) *Правило объединения*  $(A \rightarrow B) \&\& (A \rightarrow C) \Rightarrow (A \rightarrow BC)$
- 4) *Правило композиции*  $(A \rightarrow B) \&\& (C \rightarrow D) \Rightarrow (AB \rightarrow CD)$

5) *Общая теорема объединения*  $(A \rightarrow B) \&\& (C \rightarrow D) \Rightarrow (A(C \rightarrow B) \rightarrow BD)$

Два множества ФЗ  $S_1$  и  $S_2$  эквивалентны  $\Leftrightarrow$  они являются покрытиями друг друга, т. е.  $S_1+ = S_2+$ .

Множество ФЗ является неприводимым  $\Leftrightarrow$  оно обладает всеми перечисленными ниже свойствами:

- 1) Каждая ФЗ этого множества имеет одноэлементную правую часть.
- 2) Ни одна ФЗ множества не может быть устранена без изменения замыкания этого множества.
- 3) Ни один атрибут не может быть устранен из левой части любой ФЗ данного множества без изменения замыкания множества.

### 1.4.3 Общая схема процедуры нормализации

- 1) Переменную-отношение в 1НФ следует разбить на такие проекции, которые позволят исключить все функциональные зависимости, не являющиеся неприводимыми. В результате будет получен набор переменных отношений в 2НФ.
- 2) Полученные переменные-отношения в 2НФ следует разбить на такие проекции, которые позволят исключить все существующие транзитивные функциональные зависимости. В результате будет получен набор переменных-отношений в 3НФ.
- 3) Полученные переменные-отношения в 3НФ следует разбить на проекции, позволяющие исключить любые оставшиеся функциональные зависимости, в которых детерминанты не являются потенциальными ключами. В результате такого приведения будет получен набор переменных-отношений в НФБК. Замечание. Правила 1-3 могут быть объединены в одно: "Исходную переменную-отношение следует разбить на проекции, позволяющие исключить все функциональные зависимости, в которых детерминанты не являются потенциальными ключами".
- 4) Полученные переменные-отношения в НФБК следует разбить на проекции, позволяющие исключить любые многозначные зависимости, которые не являются функциональными. В результате будет получен набор переменных-отношений в 4НФ.
- 5) Полученные переменные-отношения в 4НФ следует разбить на проекции, позволяющие исключить любые зависимости соединения, которые не подразумеваются потенциальными ключами. В результате будет получен набор переменных-отношений в 5НФ.

## 1.5 Теория проектирования хранилищ данных. Основные принципы построения. ETL и ELT процессы

**Хранилище данных** – это система, в которой собраны данные из различных источников внутри компании и эти данные используются для поддержки принятия управленческих решений.

**Трехуровневая архитектура состоит из:**

- Нижний уровень содержит сервер базы данных используемый для извлечения данных из множества различных источников.
- Средний уровень содержит сервер OLAP, который преобразует данные в структуру, подходящую для анализа и сложных запросов.
- Верхний уровень — уровень клиента, содержащий инструменты, используемые для высокоуровневого анализа данных, создания отчетов и анализа данных.

Есть два хранилища данных:

- 1) *Подход Ральфа Кимбалла* основывается на важности витрин данных, которые являются хранилищами данных, облегчающие отчетность и анализ. При этом организация хранилища пространственная, где существуют «виртуальные» объекты. Коллекция витрин данных, которые могут быть пространственно разобобщены.
- 2) *Подход Билла Инмона* основывается на том, что хранилище данных является централизованным хранилищем всех корпоративных данных. При таком подходе сначала создают нормализованную модель хранилища данных, где объектами являются физически целостными. Затем создаются витрины размерных на основе модели хранилища. Это известно как нисходящий подход к хранилищу данных.

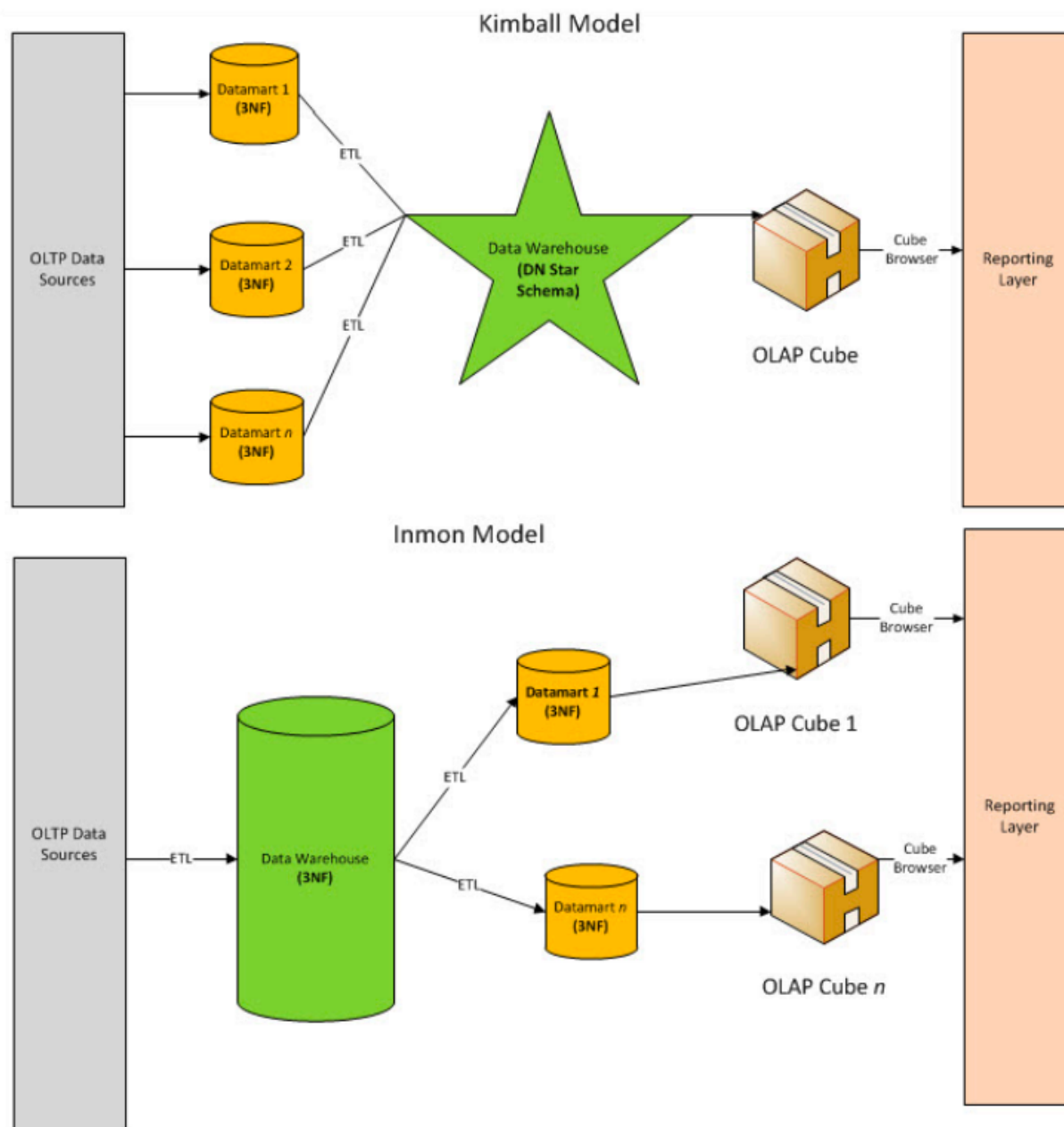


Рисунок 1 – Структуры хранилищ данных Кимбалла/Инмона

Схемы «звезда» и «снежинка» — это два способа структурировать хранилище данных.

Схема типа «звезда» имеет централизованное хранилище данных, которое хранится в таблице фактов. Схема разбивает таблицу фактов на ряд денормализованных таблиц измерений. Таблица фактов содержит агрегированные данные, которые будут использоваться для составления отчетов, а таблица измерений описывает хранимые данные.

Денормализованные проекты менее сложны, потому что данные сгруппированы. Таблица фактов использует только одну ссылку для присоединения к каждой таблице измерений. Более простая конструкция звездообразной схемы значительно упрощает написание сложных

запросов.

Схема типа «снежинка» отличается тем, что использует нормализованные данные. Нормализация означает эффективную организацию данных так, чтобы все зависимости данных были определены, и каждая таблица содержала минимум избыточности. Таким образом, отдельные таблицы измерений разветвляются на отдельные таблицы измерений.

## **ETL & ELT**

ETL и ELT — два разных способа загрузки данных в хранилище.

*ETL (Extract, Transform, Load)* сначала извлекают данные из пула источников данных. Данные хранятся во временной промежуточной базе данных. Затем выполняются операции преобразования, чтобы структурировать и преобразовать данные в подходящую форму для целевой системы хранилища данных. Затем структурированные данные загружаются в хранилище и готовы к анализу.

Инструменты ETL используются для интеграции данных, чтобы удовлетворить требованиям систем управления реляционными базами данных и/или традиционных хранилищ данных с поддержкой OLAP (online analytical processing, аналитической онлайн-обработки). Инструменты OLAP и запросы (SQL) требуют, чтобы массивы данных структурировались и стандартизировались при помощи серии преобразований, выполняемых до того, как данные попадут в хранилище.

*ELT (Extract, Load, Transform)* данные сразу же загружаются после извлечения из исходных пулов данных. Промежуточная база данных отсутствует, что означает, что данные немедленно загружаются в единый централизованный репозиторий. Данные преобразуются в системе хранилища данных для использования с инструментами бизнес-аналитики и аналитики.

## **Этапы ETL & ELT:**

- 1) Процесс загрузки данных из пула источников.
- 2) Процесс валидации (отвечает за выявление ошибок и пробелов в данных).
- 3) Процесс мэппинга данных.
- 4) Процесс агрегации данных. Этот процесс нужен из-за разности детализации данных в OLTP и OLAP системах. OLTP система может содержать несколько сумм для одного и того же набора элементов справочников, а OLAP-системы — это, по сути, полностью денормализованная таблица фактов и окружающие ее таблицы справочников.
- 5) Процесс выгрузки данных в целевую систему.

## 1.6 Транзакции. Определение, свойства и уровни изоляции транзакций. Неблагоприятные эффекты, вызванные параллельным выполнением транзакций, и способы их устранения. Управление транзакциями и способы обработки ошибок

**Транзакция** — последовательность операций, выполняемая как единое целое. (всё или ничего).

Для поддержания целостности транзакция должна обладать четырьмя свойствами АСИД:

- *Атомарность*. Транзакция либо выполняется полностью либо не выполняется вовсе.
- *Согласованность (Непротиворечивость)*. При завершении транзакции не должны быть нарушены ограничения накладываемые на данные (например constraints в БД). Согласованность подразумевает, что система будет переведена из одного корректного состояния в другое корректное
- *Изоляция*. Параллельно выполняемые транзакции не должны влиять друг на друга, например менять данные которые использует другая транзакция. Результат выполнения параллельных транзакций должен быть таким, как если бы транзакции выполнялись последовательно.
- *Долговечность (Устойчивость)*. После фиксации изменения не должны быть утеряны.

**Транзакции классифицируются по признаку определения границ:**

- 1) *Автоматические транзакции*. В этом режиме каждая инструкция T-SQL выполняется как отдельная транзакция. Если выполнение инструкции завершается успешно, происходит фиксация; в противном случае происходит откат.
- 2) *Неявные транзакции*. Если соединение работает в режиме неявных транзакции, то после фиксации или отката текущей транзакции автоматически начинает новую транзакцию. В этом режиме явно указывается только граница окончания транзакции с помощью инструкций COMMIT TRANSACTION и ROLLBACK TRANSACTION
- 3) *Явные транзакции*.

Для определения явных транзакций используются следующие инструкции:

- BEGIN TRANSACTION – задает начальную точку явной транзакции для соединения;
- COMMIT TRANSACTION или COMMIT WORK – используется для успешного завершения транзакции, если не возникла ошибка;
- ROLLBACK TRANSACTION или ROLLBACK WORK – используется для отмены транзакции, во время которой возникла ошибка.



- **SAVE TRANSACTION** – используется для установки точки сохранения или маркера внутри транзакции. Точка сохранения определяет место, к которому может возвратиться транзакция, если часть транзакции условно отменена. Если транзакция откатывается к точке сохранения, то ее выполнение должно быть продолжено до завершения с обработкой дополнительных инструкций языка T-SQL, если необходимо, и инструкции **COMMIT TRANSACTION**, либо транзакция должна быть полностью отменена откатом к началу. Для отмены всей транзакции следует использовать инструкцию **ROLLBACK TRANSACTION**; в этом случае отменяются все инструкции транзакции.

### Управлением параллельным выполнением транзакции

Когда множество пользователей одновременно пытаются модифицировать данные в базе данных, необходимо создать систему управления, которая защитила бы модификации, выполняемым одним пользователем, от негативного воздействия модификаций, сделанных другими. Выделяют два типа управления параллельным выполнением:

- *Пессимистическое управление параллельным выполнением.* При пессимистическом подходе первый пользователь захвативший данные препятствует получению данных остальным. Если конфликты редки разумно выбрать оптимистическую стратегию, так как она обеспечивает более высокий уровень параллелизма.
- *Оптимистическое управление параллельным выполнением.* При оптимистическом подходе несколько пользователей получают в свое распоряжение копии данных. Первый завершивший редактирование сохраняет изменения, остальные же должны осуществить слияние изменений. Оптимистический алгоритм позволяет конфликту произойти, но система должна восстановиться после конфликта.

В случае пессимистичного управления в СУБД не реализованы механизмы блокировки, то при одновременном чтении и изменении одних и тех же данных несколькими пользователями могут возникнуть следующие проблемы одновременного доступа:

— **Проблема последнего изменения.**

Транзакция 1	Транзакция 2
UPDATE tbl1 SET f2=f2+20 WHERE f1=1;	UPDATE tbl1 SET f2=f2+25 WHERE f1=1;

В обеих транзакциях изменяется значение поля f2, по их завершении значение поля должно быть увеличено на 45. В действительности может возникнуть следующая последовательность действий: Обе транзакции одновременно читают текущее состояние поля.

Точная физическая одновременность здесь не обязательна, достаточно, чтобы вторая по порядку операция чтения выполнялась до того, как другая транзакция запишет свой результат. Обе транзакции вычисляют новое значение поля, прибавляя, соответственно, 20 и 25 к ранее прочитанному значению. Транзакции пытаются записать результат вычислений обратно в поле f2. Поскольку физически одновременно две записи выполнить невозможно, в реальности одна из операций записи будет выполнена раньше, другая позже. При этом вторая операция записи перезапишет результат первой.

В результате значение поля f2 по завершении обеих транзакций может увеличиться не на 45, а на 20 или 25, то есть одна из изменяющих данные транзакций «пропадёт».

- **Проблема «грязного» чтения.** В транзакции 1 изменяется значение поля f2, а затем в

Транзакция 1	Транзакция 2
UPDATE tbl1 SET f2=f2+1 WHERE f1=1;  ROLLBACK WORK;	SELECT f2 FROM tbl1 WHERE f1=1;

транзакции 2 выбирается значение этого поля. После этого происходит откат транзакции 1. В результате значение, полученное второй транзакцией, будет отличаться от значения, хранимого в базе данных.

- **Проблема неповторимого чтения.** Ситуация, когда при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.

Транзакция 1	Транзакция 2
UPDATE tbl1 SET f2=f2+3 WHERE f1=1;  COMMIT;	SELECT f2 FROM tbl1 WHERE f1=1;  SELECT f2 FROM tbl1 WHERE f1=1;

В транзакции 2 выбирается значение поля f2, затем в транзакции 1 изменяется значение поля f2. При повторной попытке выбора значения из поля f2 в транзакции 2 будет получен другой результат. Эта ситуация особенно неприятна, когда данные считываются с целью их частичного изменения и обратной записи в базу данных.

- **Проблема чтения фантомов.** В транзакции 2 выполняется SQL-оператор, использующий все значения поля f2. Затем в транзакции 1 выполняется вставка новой строки,

Транзакция 1	Транзакция 2
INSERT INTO tbl1 (f1,f2) VALUES (15,20); COMMIT;	SELECT f2 FROM tbl1 WHERE f1=1;  SELECT f2 FROM tbl1 WHERE f1=1;

приводящая к тому, что повторное выполнение SQL-оператора в транзакции 2 выдаст другой результат. Такая ситуация называется чтением фантома (фантомным чтением). От не повторяющегося чтения оно отличается тем, что результат повторного обращения к данным изменился не из-за изменения/удаления самих этих данных, а из-за появления новых (фантомных) данных.

**Для решение проблем используются следующие уровни изоляции:**

- *Уровень 0* (read uncommitted) — запрещение «загрязнение» данных (повреждение данных). Этот уровень требует, чтобы изменять данные могла только одна транзакция, а другие транзакции ожидали завершение данной транзакции.
- *Уровень 1* (read committed) — запрещение «грязного» чтения. Если транзакция начала изменение данных, то никакая другая транзакция не сможет прочесть их до завершения первой.
- *Уровень 2* (repeatable read) — запрещение неповторяемого чтения. Если транзакция считывает данные, то никакая другая транзакция не сможет их изменить. Таким образом, при повторном чтении они будут находится в первоначальном состоянии.
- *Уровень 3* (snapshot, serializable)— запрещение фантомов. Если транзакция обращается к данным, то никакая другая транзакция не сможет добавить новые или удалить имеющиеся строки, которые могут быть считаны при выполнении транзакции. Реализация этого уровня блокировки выполняется путем использования блокировок диапазона ключей. Подобная блокировка накладывается не на конкретные строки таблицы, а на строки удовлетворяющие определенному логическому условию.

При этом одновременно может быть установлен только один параметр уровня изоляции, который продолжает действовать для текущего соединения до тех пор, пока не будет явно изменен. Уровни изоляции транзакции определяют тип блокировки, применяемый к операциям считывания. В любой момент транзакции можно переключиться с одного уровня изоляции на другой. Тогда для транзакции изменяется уровень изоляции, ресурсы, которые считываются после изменения, защищаются в соответствии с правилами нового уровня. Ресурсы, которые

Уровень изоляции	Грязное чтение	Неповторяющееся чтение	Фантом
read uncommitted	Да	Да	Да
read committed	Нет	Да	Да
repeatable read	Нет	Нет	Да
snapshot	Нет	Нет	Нет
serializable	Нет	Нет	Нет

считываются до изменения, остаются защищенными в соответствии с правилами предыдущего уровня.

**Способы обработки ошибок.** Если ошибка делает невозможным успешное выполнение транзакции, SQL Server автоматически выполняет ее откат и освобождает ресурсы, удерживаемые транзакцией. Если сетевое соединение клиента с SQL Server разорвано, то после того, как SQL Server получит уведомление от сети о разрыве соединения, выполняется откат всех необработанных транзакций для этого соединения. В случае сбоя клиентского приложения, выключения либо перезапуска клиентского компьютера соединение также будет разорвано, а SQL Server выполнит откат всех необработанных транзакций после получения уведомления о разрыве от сети. Если клиент выйдет из приложения, выполняется откат всех необработанных транзакций.

На случай возникновения ошибок код приложения должен содержать исправляющее действие: COMMIT или ROLLBACK. Эффективным средством для обработки ошибок, включая ошибки транзакций, является конструкция языка Transact-SQL TRY...CATCH

## **1.7 Блокировки. Определение, свойства, иерархии, гранулярность и взаимоблокировки, алгоритмы обнаружения взаимоблокировок**

### **Определение**

Блокировка — это механизм, с помощью которого компонент Database Engine синхронизирует одновременный доступ нескольких пользователей к одному фрагменту данных.

### **Свойства**

- 1) гранулярностью (или размером блокировки)
- 2) режимом (или типом блокировки)
- 3) продолжительностью

#### **1.7.1 Иерархии**

Компонент Database Engine часто получает блокировки на нескольких уровнях гранулярности одновременно, чтобы полностью защитить ресурс. Такая группа блокировок на нескольких уровнях гранулярности называется иерархией блокировки. Например, чтобы полностью защитить операцию чтения индекса, экземпляру компоненты Database Engine может потребоваться получить разделяемые блокировки на строки и намеренные разделяемые блокировки на страницы и таблицу.

#### **1.7.2 Гранулярность**

Гранулярность блокировки определяет, какой ресурс блокируется в одной попытке блокировки.

Таблица 1

Ресурс	Описание
RID	Идентификатор строки, используемый для блокировки 1 строки в куче
KEY	Блокировка строки в индексе, используемая для защиты диапазонов значений ключа в сериализуемых транзакциях.
PAGE	8КБ страница в базе данных, например страница данных или индекса.
EXTENT	Упорядоченная группа из 8 страниц, напр. стр. данных или индекса.
HOBT	Куча или сбалансированное дерево. Блокировка, защищающая индекс или кучу стр. данных в таблице, не имеющей кластеризованного индекса.
TABLE	Таблица полностью, включая все данные и индексы.
FILE	Файл базы данных.
APPLICATION	Определяемый приложением ресурс.
METADATA	Блокировки метаданных.
ALLOCATION_UNIT	Единица размещения.
DATABASE	База данных, полностью.

### 1.7.3 Взаимоблокировки

Взаимоблокировки или тупиковые ситуации (deadlocks) возникают тогда, когда одна из транзакций не может завершить свои действия, поскольку вторая транзакция заблокировала нужные ей ресурсы, а вторая в то же время ожидает освобождения ресурсов первой транзакцией. Как SQL Server обнаруживает взаимоблокировки? Обнаружение взаимоблокировки выполняется потоком диспетчера блокировок, который периодически производит поиск по всем задачам в экземпляре компонента Database Engine.

Следующие пункты описывают процесс поиска:

- 1) Значение интервала поиска по умолчанию составляет 5 секунд.
- 2) Если диспетчер блокировок находит взаимоблокировки, интервал обнаружения взаимоблокировок снижается с 5 секунд до 100 миллисекунд в зависимости от частоты взаимоблокировок.
- 3) Если поток диспетчера блокировки прекращает поиск взаимоблокировок, компонент Database Engine увеличивает интервал до 5 секунд.
- 4) Если взаимоблокировка была только что найдена, предполагается, что следующие потоки, которые должны ожидать блокировки, входят в цикл взаимоблокировки. Первая пара

элементов, ожидающих блокировки, после того как взаимоблокировка была обнаружена, запускает поиск взаимоблокировок вместо того, чтобы ожидать следующий интервал обнаружения взаимоблокировки. Например, если текущее значение интервала равно 5 секунд и была обнаружена взаимоблокировка, следующий ожидающий блокировки элемент немедленно приводит в действие детектор взаимоблокировок. Если этот ожидающий блокировки элемент является частью взаимоблокировки, она будет обнаружена немедленно, а не во время следующего поиска взаимоблокировок.

- 5) Компонент Database Engine обычно выполняет только периодическое обнаружение взаимоблокировок. Так как число взаимоблокировок, произошедших в системе, обычно мало, периодическое обнаружение взаимоблокировок помогает сократить издержки от взаимоблокировок в системе.
- 6) Если монитор блокировок запускает поиск взаимоблокировок для определенного потока, он идентифицирует ресурс, ожидаемый потоком. После этого монитор блокировок находит владельцев определенного ресурса и рекурсивно продолжает поиск взаимоблокировок для этих потоков до тех пор, пока не найдет цикл. Цикл, определенный таким способом, формирует взаимоблокировку.
- 7) После обнаружения взаимоблокировки компонент Database Engine завершает взаимоблокировку, выбрав один из потоков в качестве жертвы взаимоблокировки. Компонент Database Engine прерывает выполняемый в данный момент пакет потока, производит откат транзакции жертвы взаимоблокировки и возвращает приложению ошибку 1205. Откат транзакции жертвы взаимоблокировки снимает все блокировки, удерживаемые транзакцией. Это позволяет транзакциям потоков разблокироваться, и продолжить выполнение. Ошибка 1205 жертвы взаимоблокировки записывает в журнал ошибок сведения обо всех потоках и ресурсах, затронутых взаимоблокировкой.

## 1.8 Журнализация. Операции журнала транзакций и его логическая и физическая архитектуры. Модели восстановления. Метаданные

### 1.8.1 Типы используемых файлов

- 1) *Первичные файлы данных.* Первичный файл данных является отправной точкой базы данных. Он указывает на остальные файлы базы данных. В каждой базе данных имеется один первичный файл данных. Для имени первичного файла данных рекомендуется использовать расширение MDF
- 2) *Вторичные файлы данных.* Ко вторичным файлам данных относятся все файлы данных, за исключением первичного файла данных. Некоторые базы данных могут вообще не содержать вторичных файлов данных, тогда как другие содержат несколько вторичных файлов данных. Для имени вторичного файла данных рекомендуется использовать расширение NDF.
- 3) *Файлы журналов.* Файлы журналов содержат все сведения журналов, используемые для восстановления базы данных. В каждой базе данных должен быть, по меньшей мере, один файл журнала, но их может быть и больше. Для имен файлов журналов рекомендуется использовать расширение LDF. Журнал транзакций нельзя ни удалять, ни изменять, если только не известны возможные последствия. Кэш журнала управляется отдельно от буферного кэша для страниц данных.

### 1.8.2 Операции

- 1) *Восстановление отдельных транзакций.* Если приложение выполняет инструкцию ROLLBACK или если компонент Database Engine обнаруживает ошибку, такую как потеря связи с клиентом, записи журнала используются для отката изменений, сделанных незавершенной транзакцией.
- 2) *Восстановление всех незавершенных транзакции при запуске SQL Server.* Если на сервере, где работает SQL Server, происходит сбой, базы данных могут остаться в таком состоянии, в котором часть изменений не была переписана из буферного кэша в файлы данных, и могут быть изменения в файлах данных, совершенные незаконченными транзакциями. Когда экземпляр SQL Server будет запущен, он выполнит восстановление каждой базы данных. Будет выполнен накат каждого записанного в журнал изменения, которое, возможно, не было переписано в файл данных. Чтобы сохранить целостность базы данных, будет также произведен откат каждой незавершенной транзакции, найденной в журнале транзакций.
- 3) *Нкат восстановленно базы данных, файла, файлово группы или страницы до момента*



сбоев. После потери оборудования или сбоя диска, затрагивающего файлы базы данных, можно восстановить базу данных на момент, предшествующий сбою. Сначала восстановите последнюю полную резервную копию и последнюю дифференциальную резервную копию базы данных, затем восстановите последующую серию резервных копий журнала транзакций до момента возникновения сбоя. Агент чтения журнала следит за журналами транзакций всех баз данных, которые настроены на репликацию транзакций, и копирует отмеченные для репликации транзакции из журнала транзакций в базу данных распространителя. Решения резервного сервера, зеркальное отображение базы данных и доставка журналов в значительной степени полагаются на журнал транзакций.

- 4) Поддержка репликации транзакций.
- 5) Поддержка решений с резервными серверами.

### **1.8.3 Логическая архитектура**

На логическом уровне журнал транзакций состоит из последовательности записей. Двумя основными типами записи Log-файла являются:

- 1) код выполненной логической операции.
  - Для наката логической операции выполняется эта операция.
  - Для отката логической операции выполняется логическая операция, обратная зарегистрированной.
- 2) исходный и результирующий образ измененных данных.
  - Для наката операции применяется результирующий образ.
  - Для отката операции применяется исходный образ.

Исходный образ записи – это копия данных до выполнения операции, а результирующий образ – копия данных после ее выполнения.

### **1.8.4 Физическая архитектура**

На физическом уровне журнал транзакций состоит из одного или нескольких физических файлов. Каждый физический файл журнала разбивается на несколько виртуальных файлов журнала (VLF). VLF не имеет фиксированного размера. Не существует также и определенного числа VLF, приходящихся на один физический файл журнала. Компонент Database Engine динамически определяет размер VLF при создании или расширении файлов журнала. Компонент Database Engine стремится обслуживать небольшое число VLF. Администраторы не могут настраивать или устанавливать размеры и число VLF.

Журнал транзакций является обрабатываемым файлом. Рассмотрим пример. Пусть база данных имеет один физический файл журнала, разделенный на четыре виртуальных файла журнала. При создании базы данных логический файл журнала начинается в начале физи-

ческого файла журнала. Новые записи журнала добавляются в конце логического журнала и приближаются к концу физического файла журнала. Усечение журнала освобождает любые виртуальные журналы, все записи которых находятся перед минимальным регистрационным номером восстановления в журнале транзакций (MinLSN). MinLSN является регистрационным номером самой старой записи, которая необходима для успешного отката на уровне всей базы данных. Журнал транзакций рассматриваемой в данном примере базы данных будет выглядеть примерно так же, как на следующей иллюстрации.



Рисунок 2 – Физическая модель

Часть журнала, начинающаяся с номера MinLSN и заканчивающаяся последней записью, называется активной частью журнала, или активным журналом. Этот раздел журнала необходим для выполнения полного восстановления базы данных. Ни одна часть активного журнала не может быть усечена. Все записи журнала до номера MinLSN должны быть удалены из частей журнала.

### 1.8.5 Модель восстановления

Модель восстановления — это свойство базы данных, которое управляет процессом регистрации транзакций, определяет, требуется ли для журнала транзакций резервное копирование, а также определяет, какие типы операций восстановления доступны.

- 1) Модель полного восстановления (FULL). Обеспечивает модель обслуживания для баз данных, в которых необходима поддержка длительных транзакций. Требуются резервные копии журналов. При использовании этой модели выполняется полное протоколирование всех транзакций, и сохраняются записи журнала транзакций до момента их резервного копирования.

- 2) Модель восстановления с неполным протоколированием (BULK\_LOGGED). Эта модель восстановления обеспечивает неполное протоколирование большинства массовых операций. Она предназначена для работы только в качестве дополнения к полной модели полного восстановления. Для ряда масштабных массовых операций временное переключение на модель восстановления с неполным протоколированием повышает производительность и уменьшает место, необходимое для журналов.
- 3) Простая модель восстановления (SIMPLE). Сводит к минимуму административные затраты, связанные с журналом транзакций, поскольку его резервная копия не создается. При использовании простой модели восстановления в случае повреждения базы данных возникает риск потери значительной части результатов работы. Данные могут быть восстановлены только до момента последнего резервного копирования.

### **Восстановление данных**

- 1) База данных (полное восстановление базы данных). Вся база данных возвращается в прежнее состояние и восстанавливается, при этом база данных находится в автономном режиме во время операций возврата и восстановления.
- 2) Файл данных (восстановление файла). Файл данных или набор файлов данных возвращается в исходное состояние и восстанавливается. Во время восстановления файлов файловые группы, содержащие обрабатываемые файлы, автоматически переводятся в автономный режим на время восстановления. Любые попытки подключения и работы с недоступной файловой группой приведут к ошибке.
- 3) Страница данных (восстановление страницы). При использовании модели полного восстановления или модели восстановления с неполным протоколированием можно восстанавливать отдельные базы данных. Восстановление страниц может применяться для любой базы данных вне зависимости от числа файловых групп

### **Метаданные**

Метаданные, в общем случае, это данные о данных, информация об информации, описание контента.

Метаданные, в общем случае, это данные о данных, информация об информации, описание контента. Каждая СУБД сохраняет метаданные обо всех сущностях базы данных. Так в SQL Server с помощью инструкции CREATE можно создать 52 сущности.

В разных СУБД применяются разные названия для метаданных - системный каталог, словарь данных и др. Однако общим свойством всех современных реляционных СУБД является то, что каталог/словарь сам состоит из таблиц, а точнее - системных таблиц. В результате пользователь может обращаться к метаданным так же, как и к прикладным данным, исполь-

зуя инструкцию SELECT. Изменения же в каталоге/словаре производятся автоматически при выполнении пользователем инструкций, изменяющих состояние объектов базы данных. Системные таблицы не должны изменяться непосредственно ни одним пользователем. Например, не стоит изменять системные таблицы с помощью инструкций DELETE, UPDATE или INSERT либо с помощью пользовательских триггеров. Обращение к документированным столбцам системных таблиц разрешено. Однако многие столбцы системных таблиц не документированы. В приложениях непосредственные запросы к недокументированным столбцам применять не следует. Чтобы исключить прямой доступ к системным таблицам, пользователь «видит» не сами таблицы, а созданные на их базе представления, которые он, конечно же, не может изменять. Состав и структура каталога/словаря очень различны для различных СУБД.

**Microsoft SQL Server** предоставляет следующие коллекции системных представлений, содержащие метаданные:

- 1) Представления информационной схемы
- 2) Представления каталога
- 3) Представления совместимости
- 4) Представления репликации
- 5) Динамические административные представления и функции
- 6) Представления приложения уровня данных (DAC)

## 1.9 Безопасность и Аудит. Ключевые понятия и участники системы безопасности. Модели управления доступом

SQL Server обеспечивает защиту данных от неавторизованного доступа и от фальсификации. **Основными** функциями безопасности SQL Server являются:

- 1) проверка подлинности (аутентификация) – процедура проверки соответствия некоего лица и его учетной записи в компьютерной системе. Один из способов аутентификации состоит в задании пользовательского идентификатора, в просторечии называемого «логином» (login – регистрационное имя пользователя) и пароля – некой конфиденциальной информации, знание которой обеспечивает владение определенным ресурсом. Аутентификацию не следует путать с идентификацией. Идентификация – это установление личности самого физического лица (а не его виртуальной учетной записи, коих может быть много).
- 2) авторизация — это предоставление лицу прав на какие-то действия в системе.

**Дополнительными** функциями безопасности SQL Server являются:

- 1) шифрование,
- 2) контекстное переключение,
- 3) олицетворение,
- 4) встроенные средства управления ключами.

В основе **системы безопасности** SQL Server лежат три понятия:

- 1) участники системы безопасности (Principals);
- 2) защищаемые объекты (Securables),
- 3) система разрешений (Permissions).

### 1.9.1 Участники системы безопасности

Участники системы безопасности или принципалы – это сущности, которые могут запрашивать ресурсы SQL Server.

Принципалы могут быть иерархически упорядочены. Область влияния принципала зависит

- 1) от области его определения: Windows, SQL Server, база данных,
- 2) от того, коллективный это участник или индивидуальный. Имя входа Windows является примером индивидуального (неделимого) участника, а группа Windows — коллективного. При создании учетной записи SQL Server ей назначается идентификатор и идентификатор безопасности. В представлении каталога sys.server\_principals они отображаются в столбцах principal\_id и SID. Участники уровня Windows – это а) Имя входа домена Windows, б) Локальное имя входа Windows. Участники уровня SQL Server – это а) Имя входа SQL

Server, b) Роль сервера. Участники уровня базы данных – это а) Пользователь базы данных, b) Роль базы данных, с) Роль приложения.

### **Замечания**

- 1) Имя входа «sa» SQL Server. Имя входа sa SQL Server является участником уровня сервера. Оно создается по умолчанию при установке экземпляра. В SQL Server для имени входа sa базой данных по умолчанию будет master.
- 2) Роль базы данных public. Каждый пользователь базы данных является членом роли базы данных public. Если пользователю не были предоставлены или запрещены особые разрешения на защищаемый объект, то он наследует на него разрешения роли public.
- 3) Пользователи INFORMATION\_SCHEMA и sys. Каждая база данных включает в себя две сущности, которые отображены в представлениях каталога в виде пользователей: INFORMATION\_SCHEMA и sys. Они необходимы для работы SQL Server; эти пользователи не являются участниками и не могут быть изменены или удалены.

### **1.9.2 Защищаемые объекты**

Защищаемые объекты – это ресурсы, доступ к которым регулируется системой авторизации. Некоторые защищаемые объекты могут храниться внутри других, создавая иерархии «областей», которые сами могут защищаться. К областям защищаемых объектов относятся (а) сервер, (b) база данных и (с) схема. Далее введем следующие ограничения:

- 1) для области «сервер» ограничимся рассмотрением вопросами управления учетными записями подключения (login),
- 2) для области «база данных» ограничимся рассмотрением вопросами управления
  - учетными записями пользователей (user),
  - ролями (role),
  - ролями приложений (application role).
- 3) для области «схема» ограничимся рассмотрением вопросами управления разрешениями на
  - функции (function),
  - процедуры (stored procedure),
  - таблицы (table),
  - представления (view).

### **1.9.3 Разрешения**

У субъекта системы есть только один путь получения доступа к объектам - иметь назначенные непосредственно или опосредовано разрешения. При непосредственном управлении разрешениями они назначаются субъекту явно, а при опосредованном разрешения назначают-

ся через членство в группах, ролях или наследуются от объектов, лежащих выше по цепочке иерархии. Управление разрешениями производится путем выполнения инструкций языка DCL (Data Control Language): GRANT (разрешить), DENY (запретить) и REVOKE (отменить).

#### **1.9.4 Аудит SQL Server**

Начиная с версии SQL Server 2008 в редакции Enterprise вводится аудит SQL Server Audit – функциональность системы безопасности, которая может отслеживать практически любое действие с сервером или базой данных (выполняемое пользователями) и записывать эти действия в файловую систему или системный журнал Windows.

#### **1.9.5 Система управления на основе политик**

Одна из новых возможностей, появившаяся в SQL Server 2008, – система управления на основе политик (Policy-Based Management), которая позволяет создавать политики для обеспечения соответствия нормативам управления базой данных.

Система управления на основе политик позволяет администратору баз данных (АБД) создавать политики для управления объектами и экземплярами базы данных. Эти политики дают АБД возможность устанавливать правила создания и изменения объектов и их свойств. С помощью новой системы можно, например, создать политику уровня БД, запрещающую использование для БД свойства AutoShrink. Другой пример – политика, в соответствии с которой все имена табличных триггеров в таблице БД начинаются с tr\_.

Система управления на основе политик предусматривает использование новых терминов и понятий. Основными из них являются:

- 1) Политика (Policy) – набор условий, определенных аспектами цели управления. Другими словами, политика — это набор правил для свойств БД или серверных объектов.
- 2) Цель управления (Target) – объект, управляемый данной системой. Сюда относятся такие объекты, как экземпляр БД, база данных, таблица, хранимая процедура, триггер, индекс.
- 3) Аспект (Facet) – свойство объекта (цели управления), которое используется системой управления на основе политик. Например, имя триггера или свойство базы данных AutoShrink.
- 4) Условие (Condition) – критерий для аспектов цели управления. Например, можно создать для факта условие, по которому все имена хранимых процедур в схеме «Banking» должны начинаться с bnk\_. Кроме того, политику можно связать с определенной категорией, что позволяет осуществлять управление набором политик, привязанных к той же самой категории. Политика может принадлежать только к одной категории.

#### **1.9.6 Режим оценки политик**

Существует несколько режимов оценки политик:

- 1) По запросу (On demand) – оценку политики запускает непосредственно администратор.

- 2) При изменении: запретить (On change: prevent) – для предотвращения нарушения политики используются триггеры DDL.
- 3) При изменении: только внесение в журнал (On change: log only) – для проверки политики при изменении используются уведомления о событии.
- 4) По расписанию (On schedule) – для проверки политики на нарушения используется задание агента SQL (SQL Agent).

#### **1.9.7 Преимущества системы управления на основе политик**

Система управления на основе политик позволяет АБД в полной мере контролировать процессы, происходящие в базе данных. Администратор получает возможность реализовать принятые в компании политики на уровне БД. Политики, принятые только на бумаге, помогают определить основные принципы управления базой данных и могут служить прекрасным руководством к действию, но воплощать их в жизнь очень нелегко. Для обеспечения строгого соответствия БД принятым нормативом АБД приходится пристально следить за ее повседневным использованием и функционированием. Система управления на основе политик позволяет раз и навсегда выработать политики управления и быть уверенным в том, что они будут применяться постоянно и в полном объеме.



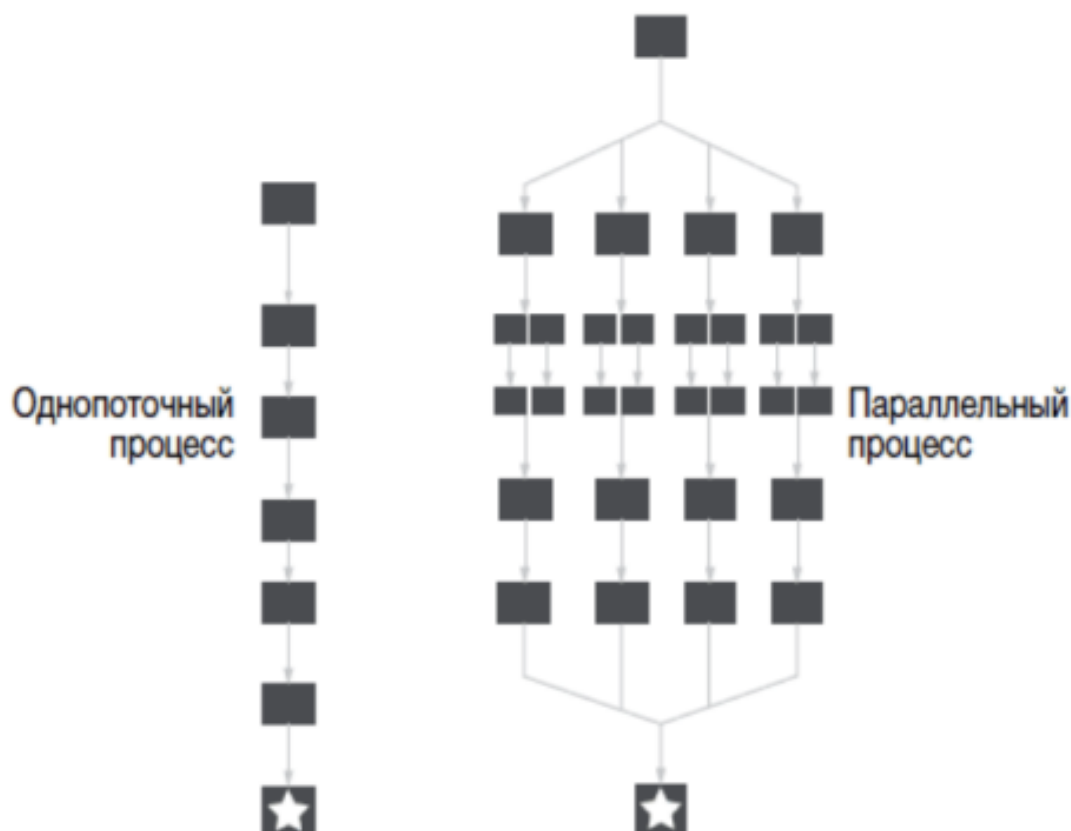
## 1.10 MPP системы. Распределенное и колоночное хранение. Распределенные вычисления, модель MapReduce. Обеспечение отказоустойчивости.

### 1.10.1 Массивно-параллельные системы обработки (massively parallel processing, MPP).

Хотя архитектуры отдельных поставщиков могут варьироваться, массивно-параллельная обработка — наиболее развитой, проверенный и широко используемый механизм хранения и анализа больших объемов данных. Так что же собой представляет массивно-параллельная архитектура и что в ней особенного? При использовании массивно-параллельной архитектуры данные разделяются на фрагменты, обрабатываемые независимыми центральными процессорами (CPU) и хранящиеся на разных носителях. Это похоже на загрузку разных фрагментов данных на несколько объединенных в сеть персональных компьютеров. Таким образом устраняется ограничение, обусловленное наличием одного центрального сервера с одним процессором и диском. Данные в массивнопараллельной системе распределяются по нескольким дискам, управляемым процессорами разных серверов.



Рисунок 3



Массивно-параллельная система вместо использования однопоточного процесса обработки данных разделяет работу на части и позволяет различным процессорам и дискам производить обработку одновременно.

Рисунок 4

При использовании МРР-систем данные не хранятся в одном месте, что облегчает восстановление в случае выхода оборудования из строя.

### 1.10.2 Аппаратная архитектура на примере VERTICA

Рассмотрим Vertica на уровне кластера. Эта СУБД обеспечивает массивно- параллельную обработку данных (МРР) в распределенной вычислительной архитектуре — «shared-nothing» — где, в принципе, любая нода готова подхватить функции любой другой ноды.

#### Основные свойства:

- 1) отсутствует единая точка отказа,
- 2) каждый узел независим и самостоятелен,
- 3) отсутствует единая для всей системы точка подключения,
- 4) узлы инфраструктуры дублируются,

5) данные на узлах кластера автоматически копируются.

Кластер без проблем линейно масштабируется. Мы просто ставим сервера в полку и подключаем их через графический интерфейс. Помимо серийных серверов, возможно развертывание на виртуальные машины. Что можно добиться **с помощью расширения**?

- 1) Увеличения объема для новых данных
- 2) Увеличение максимальной рабочей нагрузки
- 3) Повышение отказоустойчивости. Чем больше нод в кластере, тем меньше вероятность выхода кластера из строя из-за отказа, а следовательно, тем ближе мы к обеспечению доступности 24/7.

Периодически ноды нужно вынимать из кластера для обслуживания. Еще довольно распространенный кейс в крупных организациях — сервера сходят с гарантии и переходят из продуктивной в какую-нибудь тестовую среду. На их место встают новые, которые находятся на гарантии производителя. По итогам всех этих операций нужно выполнять ребалансировку. Это процесс, когда данные перераспределяются между нодами — соответственно перераспределяется рабочая нагрузка. Это требовательный к ресурсам процесс, и на кластерах с большим объемом данных он может сильно снизить производительность. Чтобы этого избежать, нужно выбрать сервисное окно — время, когда нагрузка минимальна, и в этом случае пользователи этого не заметят.

### 1.10.3 Проекции

Для понимания, как хранятся данные в Vertica, требуется разобраться с одним из основных понятий — проекцией.

Логические единицы хранения информации — это схемы, таблицы и представления.

Физические единицы — это проекции. **Проекции** бывают нескольких типов:

- 1) суперпроекции (Superprojection),
- 2) запрос-ориентированные проекции (Query-Specific Projections),
- 3) агрегированные проекции (Aggregate Projections).

При создании любой таблицы автоматически создается суперпроекция, которая содержит все колонки нашей таблицы. Если нужно ускорить какой-то из регулярных процессов, мы можем создать специальную запрос-ориентированную проекцию, которая будет содержать, допустим, 3 столбца из 10.

Для ускорения предназначен и третий тип — агрегированные проекции. Не буду вдаваться в их подклассы — это не очень интересно. Сразу хочу предупредить, что постоянно решать свои проблемы с выполнением запросов через создание новых проекций не стоит. В конечном итоге кластер начнет тормозить.

Создавая проекции, нужно оценивать, хватает ли нашим запросам суперпроекций. Если мы все-таки хотим поэкспериментировать, добавляем строго по одной новой проекции. При возникновении проблем так будет проще найти первопричину. Для больших таблиц следует создавать сегментированную проекцию. Она разбивается на сегменты, которые распределяются по нескольким нодам, что повышает отказоустойчивость и минимизирует нагрузки на одну ноду. Если таблички маленькие, то лучше делать несегментированные проекции. Они полностью копируются на каждую ноду, и производительность таким образом увеличивается. Оговорюсь: в терминах Vertica «маленькая» таблица — это примерно 1 млн строк.

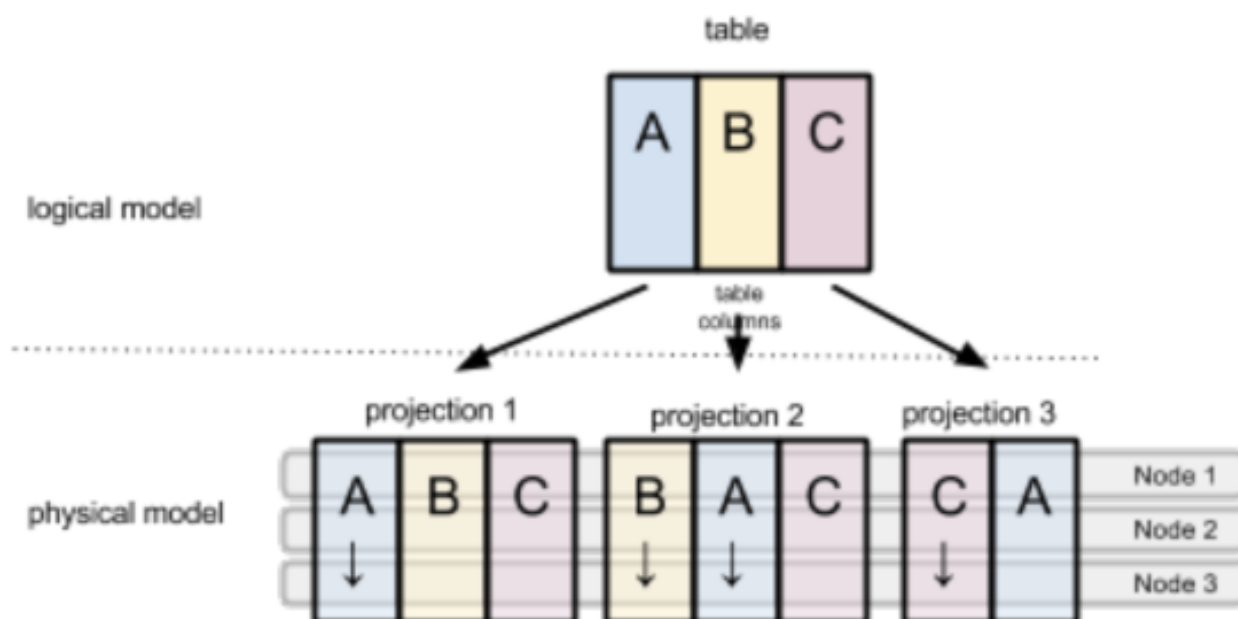


Рисунок 5

#### 1.10.4 Отказоустойчивость

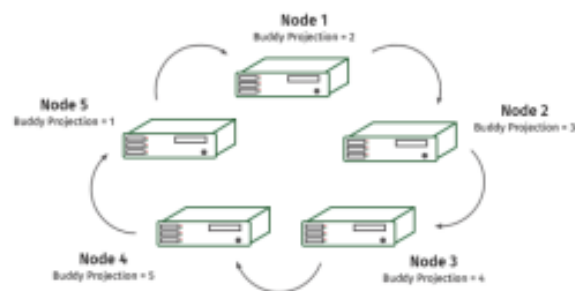
Отказоустойчивость в Vertica реализована при помощи механизма K-Safety. Он довольно простой с точки зрения описания, но сложный с точки зрения работы на уровне движка. Им можно управлять с помощью параметра K-Safety — он может иметь значение 0, 1 или 2. Этот параметр задает количество копий сегментированных проекционных данных.

Копии проекций называются **buddy projections**. Я попытался перевести это словосочетание через Яндекс-переводчик и получилось что-то вроде «проекции-кореша». Гугл предлагал варианты и интересней. Обычно данные проекции называют партнерскими или соседними, по их функциональному назначению. Это проекции, которые просто хранятся на соседних нодах и таким образом резервируются. У несегментированных проекций нет **buddy projections** — они копируются полностью.

## Как это работает?

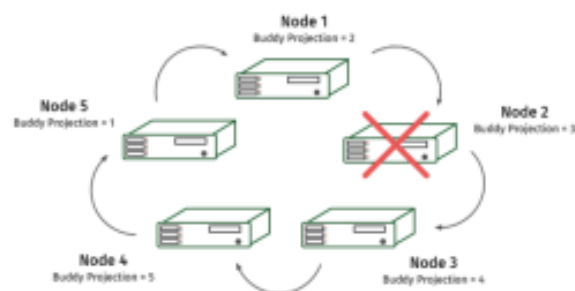
Рассмотрим кластер из пяти машин. Пусть K-safety у нас равен 1.

Ноды пронумерованы, а под ними написаны партнерские проекции, которые хранятся на них.



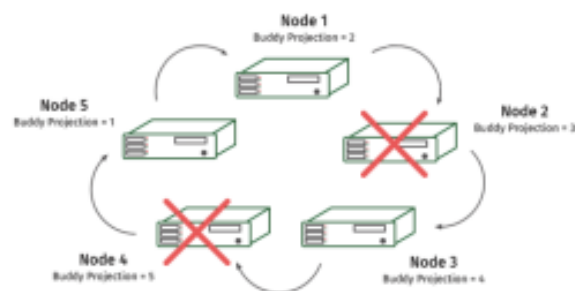
Предположим, у нас отключилась одна нода. Что будет?

Нода 1 содержит дружественную проекцию ноды 2. Поэтому на ноде 1 подрастет нагрузка, но работать кластер не перестанет.



А теперь такая ситуация:

Нода 3 содержит проекцию ноды 4, ноды 1 и 3 будут перегружены.



Усложняем задачу.

K-Safety = 2, отключим две соседние ноды. Здесь будут перегружены ноды 1 и 4 (нода 2 содержит проекцию ноды 1, а нода 3 содержит проекцию ноды 4).

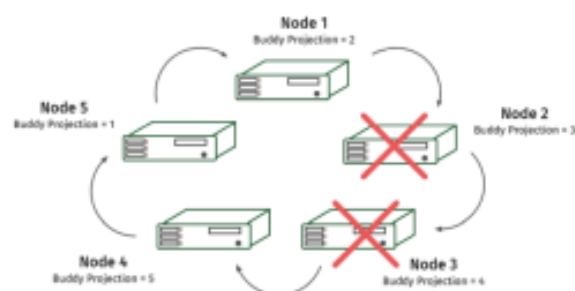


Рисунок 6

### 1.10.5 В чем выгода колоночного хранения?

Если мы читаем строки, то, например, для выполнения команды `SELECT 1,11,15 FROM table1` нам придется читать всю таблицу. Это огромный объем информации.

В данном случае колоночный подход выгоднее. Он позволяет считать только три нужных нам столбца, экономя память и время.

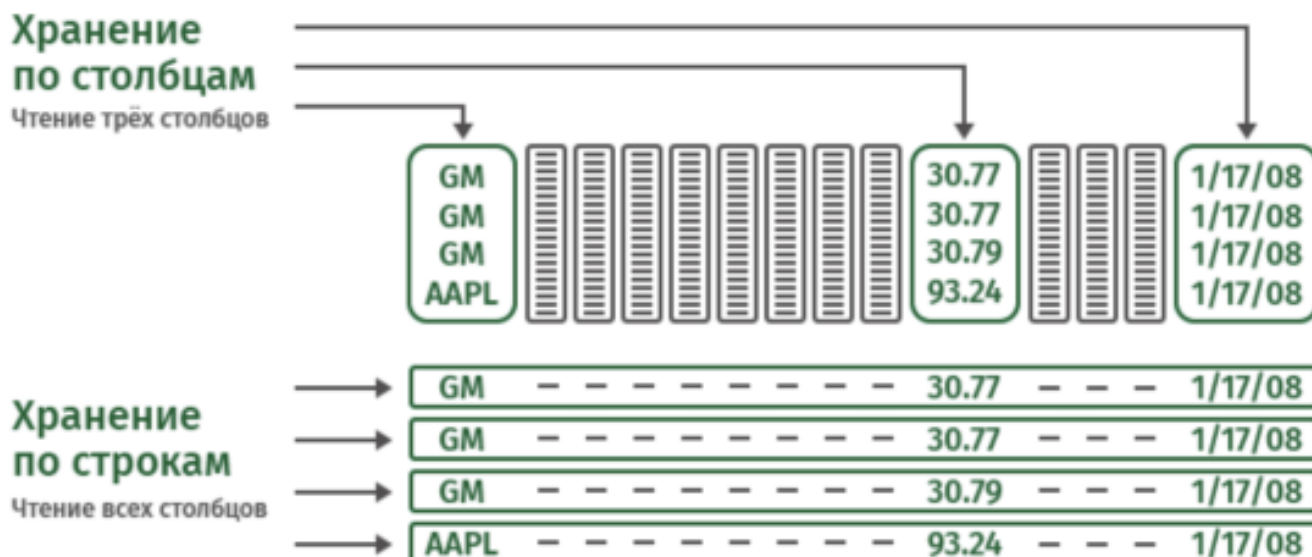


Рисунок 7

**MapReduce** — это способ агрегации больших хранилищ данных. Стадия Map выполняется на множестве серверных узлов распределенной обработки. Она обычно выполняет некую задачу на каждом распределенном серверном узле для выборки данных из узлов данных и может дополнительно преобразовывать или предварительно обрабатывать данные, когда они находятся на распределенной серверном узле. Стадия Reduce выполняется на одном или более серверных узлов конечной обработки и консолидирует все результаты от стадий Map в один конечный набор результатов, используя множество различных алгоритмов объединения.

Одно из основных преимуществ MapReduce — он обеспечивает горизонтальное масштабирование (scale out) вместо вертикального (scale up). Иначе говоря, для масштабирования вы просто добавляете обычные серверные узлы, а не приобретаете более эффективное оборудование для одного основного серверного узла. Горизонтальное масштабирование, в целом, является более дешевым и гибким выбором, поскольку используется недорогое стандартное аппаратное обеспечение, тогда как вертикальное масштабирование обычно гораздо дороже, потому что стоимость аппаратного обеспечения растет по экспоненте по мере увеличения его сложности.

### 1.11 In-Memory базы данных. Преимущества и недостатки. Примеры использования

Как это обычно бывает, у термина In-Memory DataBase нет устойчивого русского перевода, это что-то вроде «базы данных в оперативной памяти». Базы данных in-мемори хранятся в оперативной памяти сервера. Очевидно, что это обеспечивает большие преимущества в скорости, поскольку операции с данными в памяти выполняются меньшим числом инструкций процессора.

В терминах ACID базы данных in-мемори конечно **жертвуют надежностью**, ведь при обесточивании или перезагрузке в них пропадают все данные. Существуют ряд методов, которые позволяют снизить этот риск:

- 1) Снэпшоты
- 2) Логи транзакций
- 3) Использование NVRAM или NVDIMM

**Redis** – быстрое хранилище в памяти с открытым исходным кодом для структур данных «ключ-значение». Redis поставляется с набором разнообразных структур данных в памяти, что упрощает создание различных специальных приложений. Несколько фактов о ключах:

- 1) Плохая идея - хранить слишком длинные ключи.
- 2) Хорошая идея — придерживаться схемы при построении ключей: «object-type:id:field».

#### Преимущества In-Memory DataBase

- 1) Высокая производительность
- 2) Структуры данных в памяти
- 3) Универсальность и простота использования
- 4) Репликация и сохранность
- 5) Поддержка удобного языка разработки

#### Примеры использования In-Memory DataBase

- 1) Кэширование
- 2) Управление сессиями
- 3) Таблицы лидеров в режиме реального времени
- 4) Ограничение интенсивности
- 5) Очереди
- 6) Чат и обмен сообщениями

## 1.12 Инструкции языка описания данных, инструкции языка обработки данных, инструкции безопасности, инструкции управления транзакциями

### Инструкции языка описания данных

Описание данных: **Data Definition Language (DDL)** – это группа операторов описания данных. Другими словами, с помощью операторов, входящих в эту группы, мы определяем структуру базы данных и работаем с объектами этой базы, т. е. создаем, изменяем и удаляем их.

- CREATE — создает объект базы данных.
- ALTER — изменяет объект базы данных.
- DROP — удаляет объект базы данных.
- ENABLE TRIGGER — включает триггер DML, DDL или logon.
- DISABLE TRIGGER — отключает триггер.
- TRUNCATE TABLE — удаляет все строки в таблице, не записывая в журнал удаление отдельных строк.
- UPDATE STATISTICS — обновляет статистику оптимизации запросов для таблицы или индексированного представления.

### Инструкции языка обработки данных

Обработка данных: **Data Manipulation Language (DML)** – это группа операторов для манипуляции данными. С помощью этих операторов мы можем добавлять, изменять, удалять и выгружать данные из базы, т. е. манипулировать ими.

- 1) SELECT — читывает данные, удовлетворяющие заданным условиям.
- 2) INSERT — добавляет новые данные.
- 3) UPDATE — изменяет существующие данные.
- 4) DELETE — удаляет данные.
- 5) MERGE — выполняет операции вставки, обновления или удаления для целевой таблицы на основе результатов соединения с исходной таблицей. Если условие, по которому происходит объединение — истина, то можно выполнить обновление или удаление, иначе — вставку.

```
1      MERGE Основная< таблиц>
2      USING Таблица< или запрос источника>
3      ON Условия< объединения>
4      [ WHEN MATCHED [ AND Доп<. условие> ]]
5      THEN < UPDATE и л и DELETE >
6      [ WHEN NOT MATCHED [ AND Доп. условие> ]
```



```

7      THEN < INSERT > ]
8      [ WHEN NOT MATCHED BY SOURCE [ AND Доп<. условие> ]
9      THEN < UPDATE или DELETE > ] [ ... n ]
10     [ OUTPUT ]
11     ;
12

```

---

- 6) BULK INSERT — полняет импорт файла данных в таблицу или представление базы данных в формате, указанном пользователем.
- 7) READTEXT — итывает значения text, ntext или image из столбцов типа text, ntext или image начиная с указанной позиции
- 8) WRITETEXT — обновляет и заменяет все поле text, ntext или image Базы данных
- 9) UPDATETEXT — обновляет часть поля text, ntext или imag

```

1  READTEXT { table . column text_ptr offset size } [ HOLDLOCK ]
2
3  UPDATETEXT [ BULK ] { table_name . dest_column_name dest_text_ptr }
4  { NULL | insert_offset }
5  { NULL | delete_length }
6  [ WITH LOG ]
7  [ inserted_data
8  | { table_name . src_column_name src_text_ptr } ]
9
10 WRITETEXT [ BULK ]
11 { table . column text_ptr }
12 [ WITH LOG ] { data }

```

---

## Инструкции безопасности

Инструкции безопасности (ранее: доступа к данным): **Data Control Language (DCL)**.

SQL Server обеспечивает защиту данных от неавторизованного доступа и от фальсификации. Основными функциями безопасности SQL Server являются:

- проверка подлинности (аутентификация) – процедура проверки соответствия некоего лица и его учетной записи в компьютерной системе. Один из способов аутентификации состоит в задании пользовательского идентификатора, в просторечии называемого «логин» (login – регистрационное имя пользователя) и пароля – некой конфиденциальной информации, знание которой обеспечивает владение определенным ресурсом.
- авторизация – это предоставление лицу прав на какие-то действия в системе. В основе системы безопасности SQL Server лежат три понятия:

В основе системы безопасности SQL Server лежат три понятия:

- участники системы безопасности (Principals);
- защищаемые объекты (Securables);
- система разрешений (Permissions).

Участники системы безопасности или принципалы – это сущности, которые могут запрашивать ресурсы SQL Server.

Защищаемые объекты – это ресурсы, доступ к которым регулируется системой авторизации.

Инструкции безопасности:

- GRANT — предоставляет пользователю разрешения на определенные операции с объектом.
- REVOKE — отзывает ранее выданные разрешения.
- DENY — задает запрет, имеющий приоритет над разрешением.
- ADD SIGNATURE — добавляет цифровую подпись для хранимой процедуры, функции, сборки или триггера.
- OPEN MASTER KEY — открывает главный ключ в текущей базе данных.
- CLOSE MASTER KEY — закрывает главный ключ в текущей базе данных.
- OPEN SYMMETRIC KEY — расшифровывает симметричный ключ и делает его доступным для использования.
- CLOSE SYMMETRIC KEY — закрывает симметричный ключ или все симметричные ключи, открытые в текущем сеансе.
- EXECUTE AS — контекст выполнения сеанса переключается на заданное имя входа и имя пользователя.

- REVERT — переключает контекст выполнения в контекст участника, вызывавшего последнюю инструкцию EXECUTE AS.
- SETUSER — позволяет члену предопределенной роли сервера sysadmin или члену предопределенной роли базы db\_owner олицетворять другого пользователя.

### **Инструкции управления транзакциями**

**Transaction Control Language (TCL)** – группа операторов для управления транзакциями.

**Транзакция** – это команда или блок команд (инструкций), которые успешно завершаются как единое целое, при этом в базе данных все внесенные изменения фиксируются на постоянной основе или отменяются, т. е. все изменения, внесенные любой командой, входящей в транзакцию, будут отменены.

- BEGIN DISTRIBUTED TRANSACTION — запускает распределенную транзакцию, управляемую координатором распределенных транзакций.
- BEGIN TRANSACTION — отмечает начальную точку явной локальной транзакции.
- COMMIT TRANSACTION — отмечает успешное завершение явной или неявной транзакции.
- COMMIT WORK — действует так же, как и инструкция COMMIT TRANSACTION.
- ROLLBACK TRANSACTION — откатывает явные или неявные транзакции до начала или до точки сохранения транзакции.
- ROLLBACK WORK — действует так же, как и инструкция ROLLBACK TRANSACTION.
- SAVE TRANSACTION — устанавливает точку сохранения внутри транзакции.

## 1.13 Объекты базы данных: функции, процедуры, триггеры и курсоры

### 1.13.1 Функции

По поведению:

- Функция является детерминированной, если при одном и том же заданном входном значении она всегда возвращает один и тот же результат.
- Функция является недетерминированной, если она может возвращать различные значения при одном и том же заданном входном значении.

По типу возвращаемого значения:

- Скалярная функция;
- Подставляемая табличная функция;
- Многооператорная табличная функция.

**Скалярная функция** Скалярные пользовательские функции обычно используются в списке столбцов инструкции SELECT и в предложении WHERE. Табличные пользовательские функции обычно используются в предложении FROM, и их можно соединять с другими таблицами и представлениями.

Синтаксис:

```
1 CREATE FUNCTION [ имясхемы-. ] имяфункции- ( [ списокъявленияипараметров- ] )
2 RETURNS скалярныитипданных--
3 [ WITH списокопциифункции-- ]
4 [ AS ]
5 BEGIN
6 телофункции-
7 RETURN скалярноевыражение-
8 END [ ; ]
```

---

Пример:

```
1 CREATE FUNCTION dbo.AveragePrice() RETURNS smallmoney
2 WITH SCHEMABINDING AS
3 BEGIN
4 RETURN (SELECT AVG(Price) FROM dbo.P)
5 END;
6 CREATE FUNCTION dbo.PriceDifference(@Price smallmoney)
7 RETURNS smallmoney AS
8 BEGIN
9 RETURN @Price - dbo.AveragePrice()
10 END;
11 - Вызов функции
```

```
12  SELECT Pname, Price, dbo.AveragePrice() AS Average,
13  dbo.PriceDifference(Price) AS Difference
14  FROM P
15  WHERE City='Смоленск'
```

---

### Подставляемая табличная функция

Тело подставляемой табличной функции фактически состоит из единственной инструкции SELECT.

Синтаксис:

```
1  CREATE FUNCTION [ имясхемы-. ] имяфункции ( [ списокобъявленияпараметров-- ] )
2  RETURNS TABLE
3  [ WITH списокопцийфункции-- ]
4  [ AS ]
5  RETURN [ ( ) выражениевыборки- [ ) ]
6  END [ ; ]
```

---

Пример:

```
1  CREATE FUNCTION dbo.FullSPJ(@Pno int)
2  RETURNS TABLE
3  AS
4  RETURN (
5  SELECT S.Sname, P.Pname, J.Jname, SPJ.Qty
6  FROM S INNER JOIN SPJ ON S.Sno=SPJ.Sno
7  INNER JOIN P ON P.Pno=SPJ.Pno
8  INNER JOIN J ON J.Jno=SPJ.Jno
9  WHERE P.Pno = @Pno;
10 )
11 - Вызов функции
12 SELECT *
13 FROM dbo.FullSPJ(4)
```

---

### Многооператорная табличная функция

Подобно скалярным функциям, в многооператорной табличной функции команды T-SQL располагаются внутри блока BEGIN-END. Поскольку блок может содержать несколько инструкций SELECT, в предложении RETURNS необходимо явно определить таблицу, которая будет возвращаться. Поскольку оператор RETURN в многооператорной табличной функции всегда возвращает таблицу, заданную в предложении RETURNS, он должен выполняться без аргументов.

Синтаксис:

```
1 CREATE FUNCTION [ имясхемы-. ] имяфункции ( [ списокъявленияипараметров-- ] )
2 RETURNS имя@возвращаемоиперемнной--
3 TABLE определенияитаблицы-
4 [ WITH списокопцийфункции-- ]
5 [ AS ]
6 BEGIN
7 RETURN
8 END [ ; ]
```

---

Пример:

```
1 CREATE FUNCTION dbo.fnGetReports (@EmployeeID AS int )
2 RETURNS @Reports TABLE ( EmployeeID int NOT NULL,
3 ReportsToID int NULL )
4 AS
5 BEGIN
6 DECLARE @Employee int;
7
8 INSERT INTO @Reports
9 SELECT EmployeeID, ReportsTo FROM Employees
10 WHERE EmployeeID = @EmployeeID;
11
12 SELECT @Employee = MIN(EmployeeID) FROM Employees
13 WHERE ReportsTo = @EmployeeID;
14
15 WHILE @Employee IS NOT NULL
16 BEGIN
17 INSERT INTO @Reports
18 SELECT * FROM Employees;
19 END
20 RETURN
21 END;
```

---

### 1.13.2 Хранимая процедура

Хранимая процедура – именованный объект базы данных, представляющий собой набор SQL-инструкций, который компилируется один раз и хранится на сервере. Хранимые процедуры похожи на обыкновенные процедуры языков высокого уровня, у них могут быть входные и выходные параметры и локальные переменные. В хранимых процедурах могут выполняться операторы DDL, DML, TCL, FCL. Процедуры можно создавать для постоянного использования,

для временного использования в одном сеансе (локальная временная процедура), для временного использования во всех сеансах (глобальная временная процедура). Хранимые процедуры могут выполняться автоматически при запуске экземпляра SQL Server.

Синтаксис:

```
1 CREATE PROCEDURE [ имясхемы-. ] имяпроцедуры [ списокобъявленияипараметров-- ]
2 [ WITH списокопциипроцедуры-- ]
3 [ FOR REPLICATION ]
4 AS
5 телопроцедуры-
6 [ ;]
```

---

Пример:

```
1 CREATE PROCEDURE dbo.Factorial @ValIn bigint, @ValOut bigint
2 output
3 AS
4 BEGIN
5 IF @ValIn > 20 BEGIN
6 PRINT N'Входной параметр должен быть <= 20'
7 RETURN -99
8 END
9 DECLARE @WorkValIn bigint, @WorkValOut bigint
10 IF @ValIn != 1
11 BEGIN
12 SET @WorkValIn = @ValIn - 1
13 PRINT @@NESTLEVEL
14 EXEC dbo.Factorial @WorkValIn, @WorkValOut OUTPUT
15 SET @ValOut = @WorkValOut * @ValIn
16 END
17 ELSE
18 SET @ValOut = 1
19 END
```

---

Вызов хранимой процедуры:

```
1 DECLARE @FactIn int, @FactOut int
2 SET @FactIn = 8
3 EXEC dbo.Factorial @FactIn, @FactOut OUTPUT
4
5 PRINT N'Факториал ' + CONVERT(varchar(3), @FactIn) +
6 'N равен ' + CAST(@FactOut AS varchar(20))
```

---

### 1.13.3 Триггеры

Триггер - это хранимая процедура особого типа, которая выполняет одну или несколько инструкций в ответ на событие.

По типу события триггеры делятся на два класса:

- DDL-триггеры
- DML-триггеры

#### DDL-триггеры

Триггер DDL может активироваться, если выполняется такая инструкция, как ALTER SERVER CONFIGURATION, или если происходит удаление таблицы с использованием команды DROP TABLE

Пример:

```
1 CREATE TRIGGER safety
2 ON DATABASE
3 FOR DROP_TABLE , ALTER_TABLE
4 AS
5 PRINT 'You must disable Trigger "safety" to drop or alter tables!'
6 ROLLBACK;
```

---

#### DML-триггеры

Для поддержания согласованности и точности данных используются декларативные и процедурные методы. Триггеры применяются в следующих случаях:

- если использование методов декларативной целостности данных не отвечает функциональным потребностям приложения;
- если необходимо каскадное изменение через связанные таблицы в базе данных;
- если база данных денормализована и требуется способ автоматизированного обновления избыточных данных в нескольких таблицах;
- если необходимо сверить значение в одной таблице с неидентичным значением в другой таблице;
- если требуется вывод пользовательских сообщений и сложная обработка ошибок.

#### Классы DML-триггеров

Существуют два класса триггеров:

- INSTEAD OF. Триггеры этого класса выполняются в обход действию, вызывавших их срабатывание, заменяя эти действия. Например, обновление таблицы, в которой есть триггер INSTEAD OF, вызовет срабатывание этого триггера. В результате вместо оператора обновления выполняется код триггера.



- AFTER/BEFORE. Триггеры этого класса исполняются после или до действия, вызвавшего срабатывание триггера. Они считаются классом триггеров по умолчанию.

Синтаксис:

```

1  CREATE TRIGGER имятриггера_
2  ON имятаблицыилипредставления___
3  [ WITH ENCRYPTION ]
4  класстриггера_ типы() триггера_
5  [ WITH APPEND ]
6  [ NOT FOR REPLICATION ]
7  AS инструкцииsql_
8

```

---

Пример:

```

1  CREATE TRIGGER AfterUpdateSPJ
2  ON dbo.SPJ
3  AFTER UPDATE
4  AS
5  BEGIN
6  RAISERROR(N'Произошло обновление в таблице поставок',1,1)
7  END

```

---

#### 1.13.4 Курсоры

Операции в реляционной базе данных выполняются над множеством строк. Набор строк, возвращаемый инструкцией SELECT, содержит все строки, которые удовлетворяют условиям, указанным в предложении WHERE. Курсоры можно классифицировать:

- По области видимости;
- По типу;
- По способу перемещения по курсору;
- По способу распараллеливания курсора

##### **По области видимости**

По области видимости имени курсора различают:

- Локальные курсоры (LOCAL). Область курсора локальна по отношению к пакету, хранимой процедуре или триггеру, в которых этот курсор был создан. Курсор неявно освобождается после завершения выполнения пакета, хранимой процедуры или триггера, за исключением случая, когда курсор был передан параметру OUTPUT.
- Глобальные курсоры (GLOBAL). Область курсора является глобальной по отношению к соединению.

## По типу

По типу курсора различают:

- Статические курсоры (STATIC). Создается временная копия данных для использования курсором. Все запросы к курсору обращаются к указанной временной таблице в базе данных tempdb, поэтому изменения базовых таблиц не влияют на данные, возвращаемые выборками для данного курсора, а сам курсор не позволяет производить изменения.
- Динамические курсоры (DYNAMIC). Отображают все изменения данных, сделанные в строках результирующего набора при просмотре этого курсора. Значения данных, порядок, а также членство строк в каждой выборке могут меняться. Параметр выборки ABSOLUTE динамическими курсорами не поддерживается.
- Курсоры, управляемые набором ключей (KEYSET). Членство или порядок строк в курсоре не изменяются после его открытия. Набор ключей, однозначно определяющих строки, встроен в таблицу в базе данных tempdb с именем keyset.
- Быстрые последовательные курсоры (FAST\_FORWARD). Параметр FAST\_FORWARD указывает курсор FORWARD\_ONLY, READ\_ONLY, для которого включена оптимизация производительности. Параметр FAST\_FORWARD не может указываться вместе с параметрами SCROLL или FOR\_UPDATE.

## По способу перемещения по курсору

- Последовательные курсоры (FORWARD\_ONLY). Курсор может просматриваться только от первой строки к последней. Поддерживается только параметр выборки FETCH NEXT. Если параметр FORWARD\_ONLY указан без ключевых слов STATIC, KEYSET или DYNAMIC, то курсор работает как DYNAMIC.
- Курсоры прокрутки (SCROLL). Перемещение осуществляется по группе записей как вперед, так и назад. В этом случае доступны все параметры выборки (FIRST, LAST, PRIOR, NEXT, RELATIVE, ABSOLUTE). Параметр SCROLL не может указываться вместе с параметром для FAST\_FORWARD.

## По способу распараллеливания курсора

По способу распараллеливания курсоров различают:

- READ\_ONLY. Содержимое курсора можно только считывать.
- SCROLL\_LOCKS. При редактировании данной записи вами никто другой вносить в нее изменения не может. Такую блокировку прокрутки иногда еще называют «пессимистической» блокировкой.
- OPTIMISTIC. Означает отсутствие каких бы то ни было блокировок. «Оптимистическая» блокировка предполагает, что даже во время выполнения вами редактирования данных,

другие пользователи смогут к ним обращаться.

Синтаксис:

```
1  DECLARE имякурсора- CURSOR
2  [ областьвидимостиименикурсора--- ]
3  [ возможностьперемещенияпокурсору--- ]
4  [ типкурсоров- ]
5  [ опциираспараллеливаниякурсоров-- ]
6  [ выявлениеситуацииспреобразованиемтипакурсора----- ]
7  FOR инструкция_select
8  [ опция-FOR-UPDATE ]
```

---

Пример:

```
1  DECLARE @MyVariable CURSOR
2
3  DECLARE @MyCursor CURSOR
4  FOR SELECT LastName FROM
5  AdventureWorks.Person.Contact
6
7  SET @MyVariable = MyCursor;
```

---

Основой всех операций прокрутки курсора является ключевое слово `FETCH`. В качестве аргументов оператора `FETCH` могут выступать:

- `NEXT` – возвращает строку результата сразу же за текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция `FETCH NEXT` выполняет первую выборку в отношении курсора, она возвращает первую строку в результирующем наборе. `NEXT` является параметром по умолчанию выборки из курсора.
- `PRIOR` – возвращает строку результата, находящуюся непосредственно перед текущей строкой и перемещает указатель текущей строки на возвращенную строку. Если инструкция `FETCH PRIOR` выполняет первую выборку из курсора, не возвращается никакая строка и положение курсора остается перед первой строкой.
- `FIRST` – возвращает первую строку в курсоре и делает ее текущей.
- `LAST` – возвращает последнюю строку в курсоре, и делает ее текущей.

Пример:

```
1  -- Объявляем курсор
2  DECLARE CursorTest CURSOR
3  GLOBAL SCROLL STATIC FOR
4  SELECT OrderID, CustomerID
```

```
5      FROM CursorTable
6
7      -- Объявляем переменные для хранения
8      DECLARE @OrderID int
9      DECLARE @CustomerID varchar(5)
10
11     -- Откроем курсор и запросим первую запись
12     OPEN CursorTest
13     FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID --
14     Обработаем в цикле все записи курсора
15     WHILE @@FETCH_STATUS=0
16     BEGIN
17     PRINT CONVERT(varchar(5),@OrderID) + ' ' + @CustomerID
18     FETCH NEXT FROM CursorTest INTO @OrderID, @CustomerID
19     END
20
21     CLOSE CursorTest
```

---

## 1.14 Оптимизация запроса: индексы, партиционирование, сегментирование

### 1.14.1 Индексы

(материал на основе SQL Server)

Индекс – это объект базы данных, обеспечивающий дополнительные способы быстрого поиска и извлечения данных. Индекс может создаваться на одном или нескольких столбцах. Это означает, что индексы бывают простыми и составными. Если в таблице нет индекса, то поиск нужных строк выполняется простым сканированием по всей таблице. При наличии индекса время поиска нужных строк можно существенно уменьшить.

К недостаткам индексов следует отнести:

- дополнительное место на диске и в оперативной памяти,
- замедляются операции вставки, обновления и удаления записей.

В SQL Server (и, наверное, многих других СУБД) индексы хранятся в виде сбалансированных деревьев. Представление индекса в виде сбалансированного дерева означает, что стоимость поиска любой строки остается относительно постоянной, независимо от того, где находится эта строка.

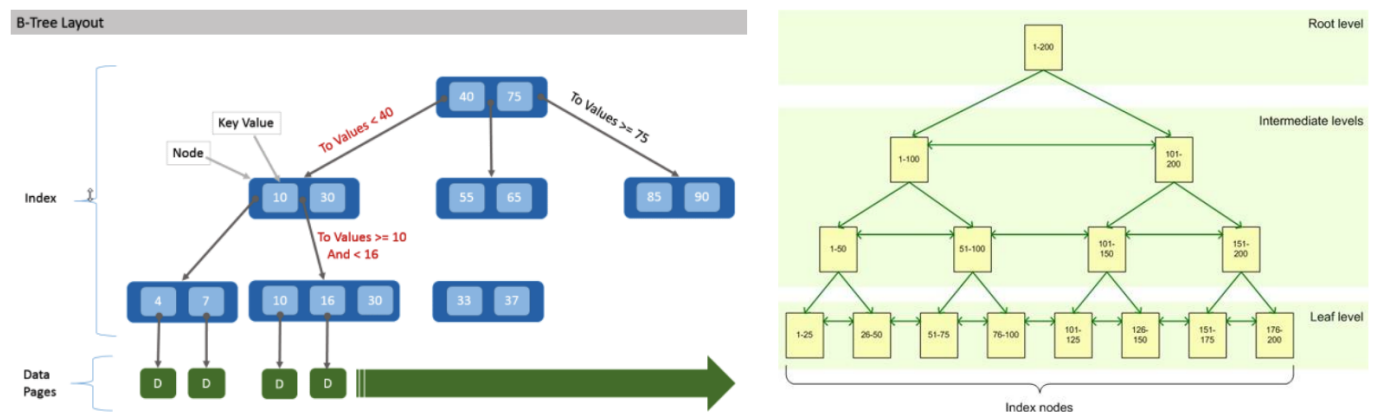


Рисунок 8 – B-Tree

Существует два типа индексов:

- Кластерные индексы
- Некластерные индексы, которые включают:
  - индексы на основе кучи;
  - индексы на основе кластерных таблиц.

#### Кластерные индексы

В кластерном индексе таблица представляет собой часть индекса, или индекс представляет собой часть таблицы в зависимости от вашей точки зрения. Листовой узел кластерного индекса – это страница таблицы с данными. Поскольку сами данные таблицы являются частью индекса,

для таблицы может быть создан только один кластерный индекс. Кластерный индекс является уникальным индексом по определению.

### **Некластерные индексы на основе кучи**

В листьях некластерного индекса на основе кучи хранятся указатели на строки данных. Указатель строится на основе идентификатора файла (ID), номера страницы и номера строки на странице. Весь указатель целиком называется идентификатором строки (RID).

### **Некластерные индексы, основанные на кластерных таблицах**

В листьях некластерного индекса, основанного на кластерных таблицах, хранятся указатели на корневые узлы кластерных индексов. Поиск в таком индексе состоит из двух этапов:

- поиск в некластерном индексе;
- поиск в кластерном индексе.

### **Создание индекса**

В SQL Server индексы создаются при помощи команды **CREATE INDEX**.

Помимо этого, индексы создаются при добавлении некоторых ограничений. Такой тип индексов часто называют «связанными индексами». Связанные индексы создаются при добавлении одного из следующих двух типов ограничений:

- ограничения первичного ключа (PRIMARY KEY);
- ограничения уникальности (UNIQUE).

### **1.14.2 Партиционирование**

*(материал на основе PostgreSQL)*

Партиционирование – это метод разделения больших (исходя из количества записей, а не столбцов) таблиц на много маленьких.

Для произведения партиционирования нужно решить, каким будет ключ партиционирования – другими словами, по какому алгоритму будут выбираться партии. Есть пара наиболее очевидных:

- 1) партиционирование по дате – например, выбирать партии, основываясь на годе, в котором пользователь был создан.

*Достоинства:*

- легко понять;
- количество строк в данной таблице будет достаточно стабильным.

*Недостатки:*

- требует поддержки – время от времени нам придётся добавлять новые партии;
- поиск по имени пользователя или id потребует сканирования всех партий;

- 2) партиционирование по диапазону идентификаторов – например, первый миллион пользователей, второй миллион пользователей, и так далее

*Достоинства:*

- легко понять;
- количество строк в данной таблице будет абсолютно стабильным.

*Недостатки:*

- требует поддержки – время от времени нам придётся добавлять новые партиции;
- поиск по имени пользователя потребует сканирования всех партиций;

- 3) партиционирование по чему-нибудь другому – например, по первой букве имени пользователя.

*Достоинства:*

- легко понять;
- никакой поддержки — есть строго определенный набор партиций и нам никогда не придется добавлять новые;

*Недостатки:*

- количество строк в партициях будет стабильно расти;
- в некоторых партициях будет существенно больше строк, чем в других (больше людей с никами, начинающимися на «t\*», чем на «y\*»);
- поиск по id потребует сканирования всех партиций;

- 4) есть еще пара других, не так часто используемых вариантов, вроде «партиционирования по хэшу от имени пользователя».

### Пример

```
1 CREATE TABLE measurement (  
2   city_id int not null,  
3   logdate date not null,  
4   peaktemp int,  
5   unitsales int  
6 ) PARTITION BY RANGE (logdate);
```

---

### 1.14.3 Сегментирование

*(материал на основе Vertica)*

Для распределения данных по кластерам используется их сегментация (Segmentation), а точнее сегментация проекций (Projection) в которых они находятся. Задача разработчика — подобрать такой список полей и/или такую функцию (например, хэш-функцию), благодаря которым данные равномерно распределятся по нодам кластера. HP Vertica рекомендует исполь-

зовать встроенные функции HASH и MODULARHASH для этих целей.

*Прим.* Для справки: *K-Safety* — критерий отказоустойчивости БД, определяющий без какого кол-ва узлов БД сможет функционировать корректно. Он может быть равен 0, 1 или 2.

Для обеспечения  $K\text{-Safety} > 0$  создаются сообщные проекции (Buddy Projection). Проекции называются сообщными, если они имеют в себе одинаковые наборы полей и одинаковое выражение сегментации, но хранятся на разных нодах. Сообщные проекции позволяют создать те самые реплики, которые позволяют работать БД в выбранном режиме *K-Safety*.

## Пример

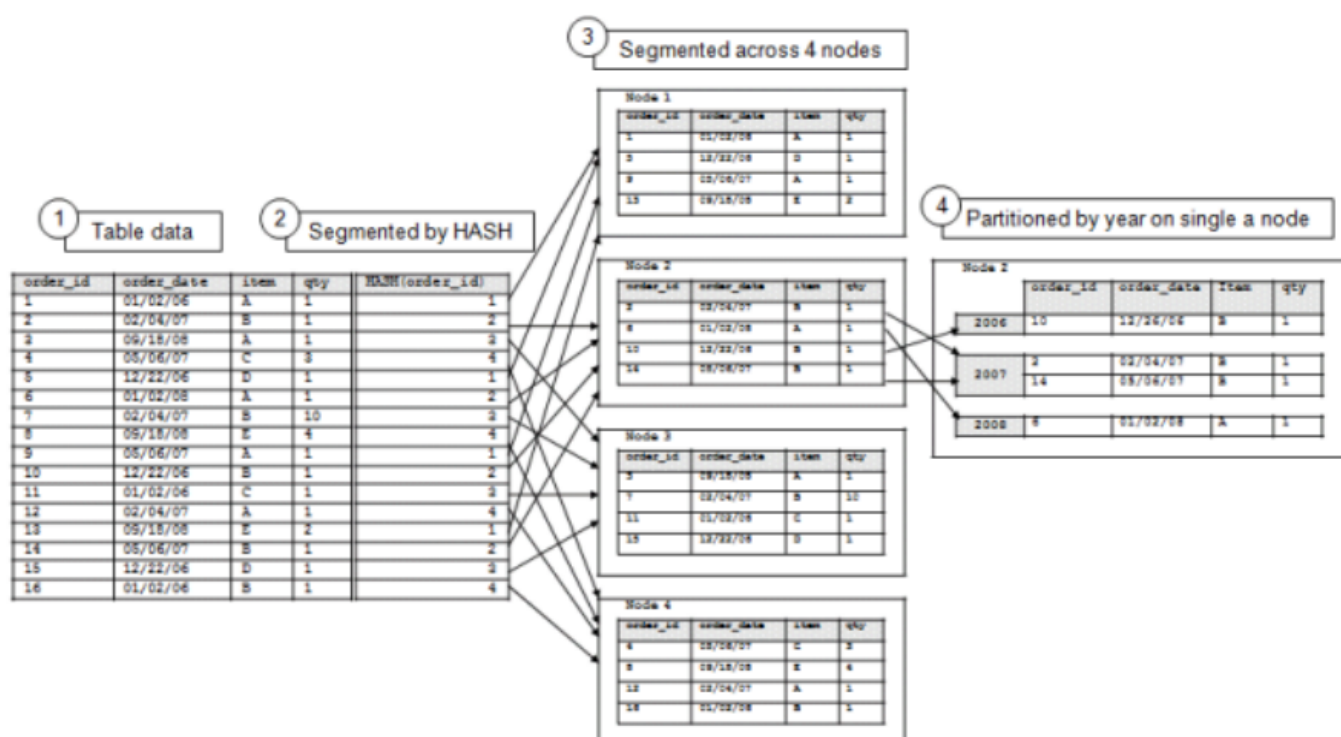


Рисунок 9 – Сегментирование и партиционирование вместе



### 1.15 План запроса. Этапы выполнения запроса

Унифицированный сервер баз данных, имеющий единый движок – storage engine. Постгрес использует клиент-серверную модель. При подключении к серверу клиент соединяется с процессом postmaster. В задачи этого процесса входит:

- 1) Порождение других процессов
- 2) Присмотр за созданными процессами.

Таким образом, postmaster порождает серверный процесс и дальше клиент работает уже с ним. На каждое соединение создается по серверному процессу, поэтому при большом числе соединений следует использовать пул (например, с помощью расширения pgbouncer). Postmaster также порождает ряд служебных процессов. Дерево процессов можно увидеть с помощью команды ps fax.

У экземпляра СУБД имеется общая для всех серверных процессов память. Большую ее часть занимает буферный кэш (shared buffers), необходимый для ускорения работы с данными на диске. Обращение к дискам происходит через операционную систему (которая тоже кэширует данные в оперативной памяти).

PostgreSQL полностью полагается на операционную систему и сам не управляет устройствами. В частности, он считает, что вызов fsync() гарантирует попадание данных из памяти на диск.

Кроме буферного кэша в общей памяти находится информация о блокировках и многое другое; через нее же серверные процессы общаются друг с другом. У каждого серверного процесса есть своя локальная память. В ней находится кэш каталога (часто используемая информация о базе данных), планы запросов, рабочее пространство для выполнения запросов и другое. Для работы с такими клиентскими процессами сервер использует семафоры.

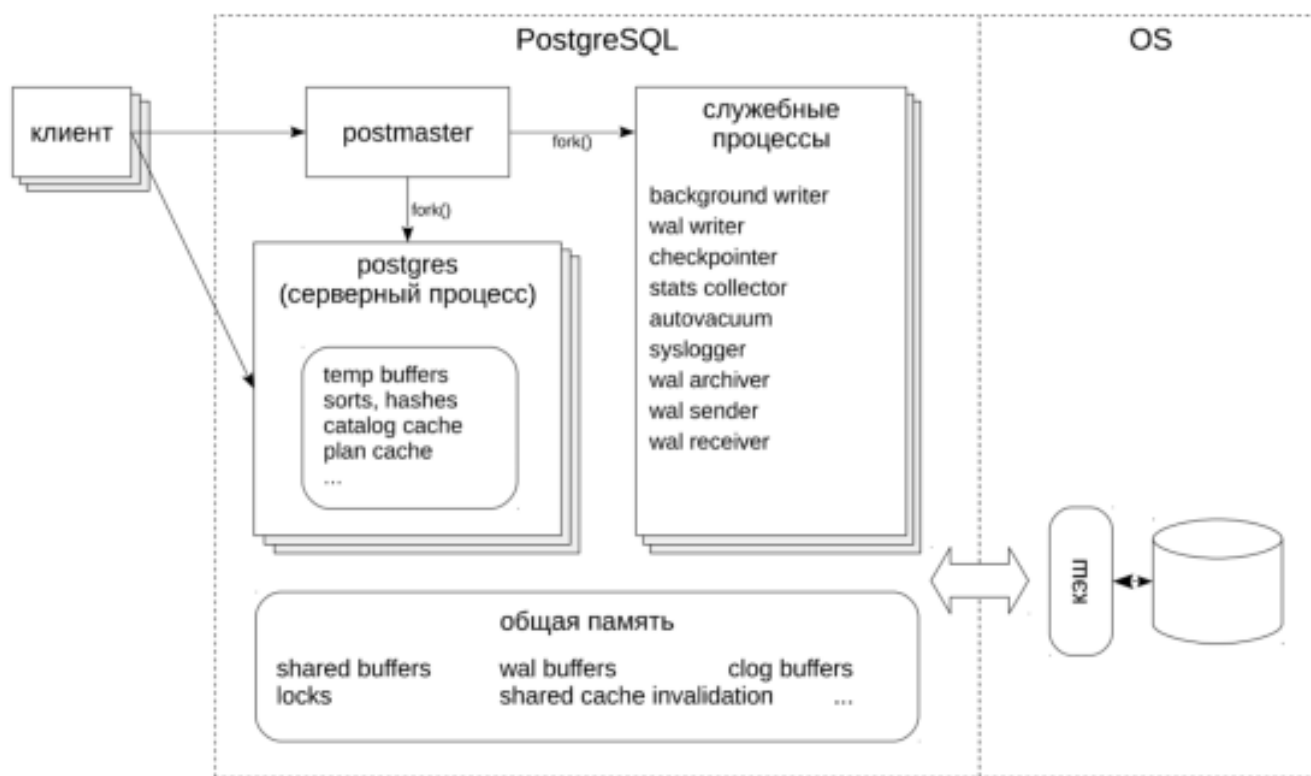


Рисунок 10

### 1.15.1 Пять стадий выполнения запроса

Клиентский запрос проходит следующие стадии:

- 1) Прикладная программа устанавливает подключение к серверу PostgreSQL.
- 2) На этапе разбора запроса сервер выполняет синтаксическую проверку запроса, переданного прикладной программой, и создаёт дерево запроса. Дерево разбора состоит из узлов двух типов:
  - Атомы - лексические элементы следующих типов: ключевые слова (например, SELECT); имена атрибутов или отношений; константы; скобки; операторы (например, + или >); и др.
  - Синтаксические категории - имена семейств, представляющих часть запроса. Заключаются в угловые скобки: <SFW>, <Condition>
- 3) Система правил принимает дерево запроса, созданное на стадии разбора, и ищет в системных каталогах правила для применения к этому дереву. Обнаружив подходящие правила, она выполняет преобразования, заданные в теле правил. Одно из применений системы правил заключается в реализации представлений. Когда выполняется запрос к представлению (т. е. виртуальной таблице), система правил преобразует запрос пользователя в запрос, обращающийся не к представлению, а к базовым таблицам из определения пред-

ставления.

- 4) Планировщик/оптимизатор принимает дерево запроса (возможно, переписанное) и создаёт план запроса, который будет передан исполнителю. Он выбирает план, сначала рассматривая все возможные варианты получения одного и того же результата. Например, если для обрабатываемого отношения создан индекс, прочитать отношение можно двумя способами. Во-первых, можно выполнить простое последовательное сканирование, а во-вторых, можно использовать индекс. Затем оценивается стоимость каждого варианта и выбирается самый дешёвый. Затем выбранный вариант разворачивается в полноценный план, который сможет использовать исполнитель.
- 5) Исполнитель рекурсивно проходит по дереву плана и получает строки тем способом, который указан в плане. Он сканирует отношения, обращаясь к системе хранения, выполняет сортировку и соединения, вычисляет условия фильтра и, наконец, возвращает полученные строки.

### 1.15.2 План запроса

План выполнения показывает, как будут сканироваться таблицы, затрагиваемые оператором — просто последовательно, по индексу и т. д. — а если запрос связывает несколько таблиц, какой алгоритм соединения будет выбран для объединения считанных из них строк.

Наибольший интерес в выводимой информации представляет ожидаемая стоимость выполнения оператора, которая показывает, сколько, по мнению планировщика, будет выполняться этот оператор (это значение измеряется в единицах стоимости, которые не имеют точного определения, но обычно это обращение к странице на диске). Фактически выводятся два числа: стоимость запуска до выдачи первой строки и общая стоимость выдачи всех строк. Для большинства запросов важна общая стоимость, но в таких контекстах, как подзапрос в EXISTS, планировщик будет минимизировать стоимость запуска, а не общую стоимость (так как исполнение запроса всё равно завершится сразу после получения одной строки). Кроме того, если количество возвращаемых строк ограничивается предложением LIMIT, планировщик интерполирует стоимость между двумя этими числами, выбирая наиболее выгодный план.

### 1.15.3 Как вызвать план запроса в postgres:

EXPLAIN — показать план выполнения оператора

Примеры для демонстрации плана запроса:











<pre>explain select * from dbspj.spj s</pre>	 QUERY PLAN  Seq Scan on spj s (cost=0.00..28.50 rows=1850 width=16)
<pre>explain SELECT Sname FROM dbSPJ.S s JOIN dbSPJ.SPJ spj ON s.Sno = spj.Sno WHERE Qty &gt; 500;</pre> <pre>explain SELECT Sname FROM dbSPJ.S s JOIN dbSPJ.SPJ spj ON s.Sno = spj.Sno AND Qty &gt; 500;</pre>	 QUERY PLAN  Hash Join (cost=23.05..120.34 rows=1789 width=58) Hash Cond: (spj.sno = s.sno) -> Seq Scan on spj (cost=0.00..33.12 rows=617 width=4) Filter: (qty > 500) -> Hash (cost=15.80..15.80 rows=580 width=62) -> Seq Scan on s (cost=0.00..15.80 rows=580 width=6)  QUERY PLAN  Hash Join (cost=23.05..120.34 rows=1789 width=58) Hash Cond: (spj.sno = s.sno) -> Seq Scan on spj (cost=0.00..33.12 rows=617 width=4) Filter: (qty > 500) -> Hash (cost=15.80..15.80 rows=580 width=62) -> Seq Scan on s (cost=0.00..15.80 rows=580 width=6)
<pre>explain SELECT sp.pno, sp.jno, m.qty FROM dbspj.spj sp JOIN ( SELECT spj.jno, MAX(qty) AS qty FROM dbspj.spj spj GROUP BY spj.jno) m ON sp.jno = m.jno AND sp.qty = m.qty</pre>	 QUERY PLAN  Hash Join (cost=44.75..82.96 rows=9 width=12) Hash Cond: ((sp.jno = m.jno) AND (sp.qty = m.qty)) -> Seq Scan on spj sp (cost=0.00..28.50 rows=1850 width=16) -> Hash (cost=41.75..41.75 rows=200 width=8) -> Subquery Scan on m (cost=37.75..41.75 rows=200 width=8) -> HashAggregate (cost=37.75..39.75 rows=200 width=8) Group Key: spj.jno -> Seq Scan on spj (cost=0.00..28.50 rows=1850 width=16)
<pre>explain SELECT pno, jno, qty FROM (SELECT pno, jno, qty, ROW_NUMBER() OVER(PARTITION BY jno ORDER BY qty DESC) AS rn FROM dbspj.spj spj) m WHERE m.rn = 1</pre>	 QUERY PLAN  Subquery Scan on m (cost=128.89..189.02 rows=9 width=12) Filter: (m.rn = 1) -> WindowAgg (cost=128.89..165.89 rows=1850 width=24) -> Sort (cost=128.89..133.52 rows=1850 width=12) Sort Key: spj.jno, spj.qty DESC -> Seq Scan on spj (cost=0.00..28.50 rows=1850 width=16)

Рисунок 11