*Базы данных (часть 2)

Кореньков Владимир Васильевич

зав. кафедры «Распределенные информационно-вычислительные системы»,

директор Лаборатории информационных технологий ОИЯИ

Основные темы лекций по курсу СУБД

1. Основные понятия баз данных. Этапы развития СУБД. Требования к системам управления базами данных.

2. Архитектура баз данных. Логическая и физическая независимость данных. Схема прохождения запросов к БД. Режимы работы с базой данных. Схема прохождения запроса к БД. Классификация моделей данных. Архитектура и модели "клиент-сервер" в технологии БД.

3. Реляционная модель БД. Таблица, кортеж, атрибут, домен, первичный ключ, внешний ключ. Основные достоинства реляционной модели. Фундаментальные свойства отношений. Обеспечение целостности данных.

4. Операторы реляционной алгебры. Понятия полной и транзитивной функциональной зависимости. Нормализация, нормальные формы.

5. Проектирование баз данных. Семантические модели данных. ER - модель (Entity-Relationship, Сущность-Связи). Этапы проектирования баз данных.

6. Язык SQL, его структура, стандарты, история развития. Подмножество языка DML: операторы SELECT, INSERT, UPDATE, DELETE.

7. Подмножество языка DDL: операторы CREATE, ALTER, DROP. Поддержка ссылочной целостности данных. Представления, их значение. Обновляемые представления.

8. Объектные и системные привилегии. Операторы GRANT, REVOKE. Роли. Транзакции. Операторы управления транзакциями: COMMIT, ROLLBACK, SAVEPOINT. Журнал транзакций.

* Темы лекций

1. Язык PL/SQL, его структура, основные операторы.
2. Курсоры, операторы работы с курсором, оператор SELECT INTO.
3. Процедуры, функции, пакеты.
4. Триггеры, их основные свойства и значение.
5. Параллельные архитектуры БД
6. Распределенные базы данных, фрагментация, тиражирование.
7. Доступ к базам данных. Архитектура ODBC.
8. Объектно-ориентированные базы данных
9. ООСУБД, преимущества, недостатки, реализации
10. Этапы развития СУБД ORACLE (ORACLE 8i, 9i, 10G, 11G)
11. Классификация NoSQL, теорема **CAP,** свойство ***BASE***
12. Масштабируемость, репликации, шардинг, Map/Reduce
13. Особенности и реализации хранилищ типа «ключ-значение»
14. Особенности и реализации колоночных хранилищ
15. Особенности и реализации документно-ориентированных хранилищ
16. Особенности и реализации хранилищ графов
17. Новые архитектуры баз данных.
18. Облачные СУБД, платформа Hadoop

Литература

* Дейт К. Введение в системы баз данных. – 8 изд., Вильямс, 2005
* Кузнецов С.Д. Базы данных. Модели и языки – М: Бином-Пресс, 2008
* Малыхина М.П. Базы данных: основы, проектирование, использование – Спб: БХВ-Петербург, 2006
* Маркин А.В. Построение запросов и программирование на SQL - М.: Диалог-МИФИ, 2008.
* Урман С. Oracle 10g: Программирование на языке PL/SQL - М.: Лори, 2007
* Т. Конноли, К. Бегг, А. Страхан «Базы данных. Проектирование, реализация и сопровождение. Теория и практика». – М: «Вильямс», 2000
* Генник Д. Справочник по SQL. – М.: Питер, 2004
* Прайс Д. Oracle Database 11g. SQL и PL/SQL - Лори, 2012 г.
* А. Саймон Стратегические технологии баз данных. – М., Финансы и статистика, 2000
* М.Р. Когаловский «Энциклопедия технологий баз данных». – М:, Финансы и статистика, 2002
* www.osp.ruwww.osp.ru, www.citforum.ru

PL/SQL (Procedural Language) — процедурное расширение языка SQL

PL/SQL - Procedural Language. Как видно из его названия, PL/SQL расширяет возможности SQL, добавляя в него конструкции процедурных языков, такие как:

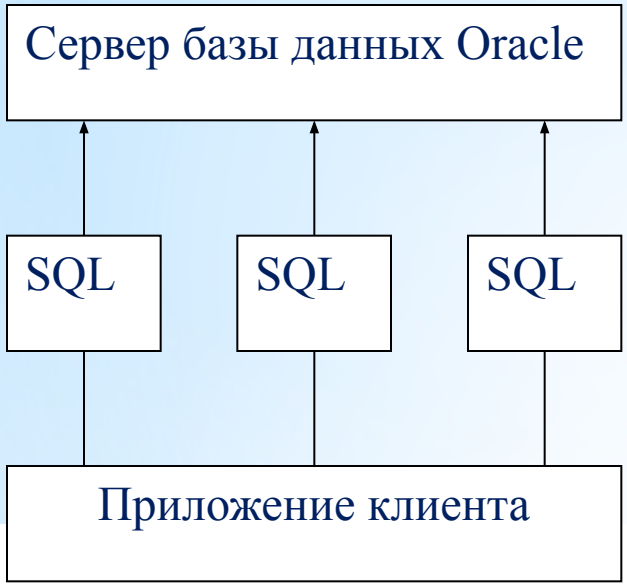
- ✔ Переменные и типы (как предварительно определенные, так и определяемые пользователем).
- ✔ Управляющие структуры, такие как операторы и циклы IF-THEN-ELSE.
- ✔ Процедуры и функции.
- ✔ Объектные типы и методы (PL/SQL версии 8 и выше).

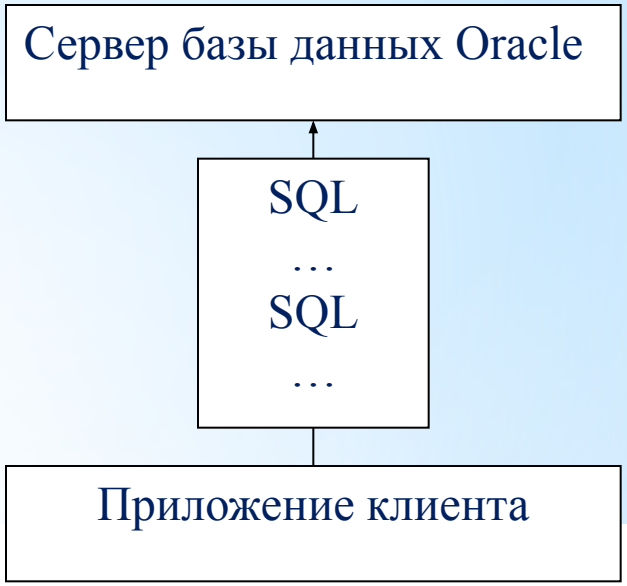
PL/SQL — полноценный язык программирования, не являющийся самостоятельным продуктом. Это технология, включающая механизм, выполняющий блоки PL/SQL (PL/SQL engine). Механизм может быть встроен в ядро СУБД, или же в любое инструментальное средство Oracle.

PL/SQL уникален тем, что соединяет гибкость SQL с мощью и способностью к конфигурированию языка 3GL. В нем имеются как необходимые процедурные конструкции, так и возможность обращения к базе данных. Таким образом, этот язык программирования является надежным, эффективным языком, хорошо подходящим для разработки сложных приложений.

Основное отличие PL/SQL от других процедурных языков заключается во встроенном механизме обработки курсоров, позволяющем выполнять операторы непроцедурного языка запросов SQL из PL/SQL-программы.

* Модель клиент — сервер





Использование SQL

Использование PL/SQL

PL/SQL в среде клиент /сервер

Многие приложения для работы с базами данных создаются с использованием модели клиент /сервер.
Сама программа размещается на компьютере клиента и посылает запросы на получение информации серверу базы данных. Запросы инициируются при помощи SQL, что приводит к наличию в сети большого числа посылок — по одной на каждый SQL-оператор.

Несколько SQL-операторов могут быть объединены в единый блок PL/SQL и посланы серверу как единое целое.

В результате сетевой трафик снижается, а приложение функционирует намного быстрее.

*Блок PL/SQL

Базовой единицей PL/SQL является блок (block). Все программы PL/SQL состоят из блоков, которые могут быть вложены один в другой.

Блок имеет следующую структуру:

DECLARE

<Раздел объявлений переменных, типов, курсоров и логических подпрограмм PL/SQL >

BEGIN

<Выполняемый раздел – процедурные и SQL- операторы. Это основной раздел блока и единственный, являющийся обязательным. >

EXCEPTION

<Раздел исключительных ситуаций – операторы обработки ошибок. >

END;

Допустимы следующие виды блоков:

- ***Анонимные (непоименованные) блоки*** создаются, как правило, динамически и выполняются только один раз
- ***Именованные блоки*** – это анонимные блоки с метками, дающими блокам имена. Они также создаются как правило, динамически и выполняются только один раз.
- ***Подпрограммы*** – это процедуры, модули и функции, хранимые в базе данных. Эти блоки, как правило, не изменяются и выполняются многократно явным образом посредством вызова процедуры, модуля или функции.
- ***Триггеры*** – это именованные блоки, которые также хранятся в базе данных. Они тоже, как правило, не изменяются и выполняются многократно неявным образом при наступлении соответствующих событий. Событием, вызывающим активизацию триггера, является оператор языка DML, выполняемый над некоторой таблицей базы данных.

Замечание *При создании процедуры ключевое слово DECLARE необязательно. Более того, его использование будет ошибкой. Однако DECLARE требуется при создании триггера.*

*Лексические единицы

Набор символов PL/SQL

При работе с PL/SQL допускается использование символов из определенного набора знаков. В этот набор входят почти все символы, которые можно ввести с клавиатуры. Однако существуют ограничения на применение ряда символов в некоторых конкретных ситуациях.

Набор символов, который можно использовать при программировании на PL/SQL:

- Все прописные и строчные буквы
- Цифры от 0 до 9
- Знаки ( ) + √ * / > < = ! ~ ; : . ' @ % , " # $ ^ & _ { } ? [ ]

1. Арифметические операторы

Оператор	Операция	Оператор	Операция
+	сложение	/	деление
-	вычитание	**	возведение в степень
*	умножение		

2. Операторы сравнения

Оператор	Операция	Оператор	Операция
<>	не равно	<	меньше
!=	не равно	>	больше
^=	не равно	=	равно

* Идентификаторы

Идентификаторы используются для именования переменных, курсоров, типов и подпрограмм. При выборе идентификаторов следует руководствоваться следующими правилами:

- **Идентификатор должен начинаться с буквы (A-Z).**
- **За первой буквой переменной может следовать одна или несколько букв, цифр (0-9) или специальных символов $, # или _ .**
- **Длина идентификатора не может превышать 30 символов.**
- **Идентификатор не может содержать пробелы.**

Пример:

currentcustomer CHAR(15); -- переменная

Константы

При объявлении константы указывается CONSTANT , а после идентификатора типа – оператор присваивания и значение константы.

discont CONSTANT REAL := 0.1; -- константа

* Типы данных

Тип	Подтип
NUMBER	DECIMAL, REAL, FLOAT, NUMERIC
(precision, scale)	INTEGER, SMALLINT,
CHAR (length)	
VARCHAR2 (length)	
DATE	
BOOLEAN	
RECORD	- составной тип данных
TABLE	- составной тип данных

Пример:

```
DECLARE
    TYPE  orderrecordtype IS RECORD
          ( id number(5,0) NOT NULL :=0,
            customerid NUMBER(5,0) NOT NULL :=0,
            orderdate DATE NOT NULL :=SYSDATE);
```

* Записи PL/SQL

Записи (records) PL/SQL аналогичны структурам языка C. С помощью записи можно работать с несколькими отдельными, но связанными переменными как с одной программной единицей.

Объявим тип записи для хранения информации о студентах.

```
DECLARE
TYPE t_StudentRecord1 IS  RECORD (
        StudentID NUMBER(5);
        FirstName  VARCHAR2(20);
        LastName  VARCHAR2(20));
TYPE t_StudentRecord2 IS  RECORD (
        StudentID NUMBER(5);
        FirstName  VARCHAR2(20);
        LastName  VARCHAR2(20));
/* объявим переменные с этими типами*/
vStudentInfo1   t_StudentRecord1 ;
vStudentInfo2   t_StudentRecord2 ;
```

Чтобы присвоить одной записи значение другой они должны быть одного типа.

Хотя записи имеют одинаковые имена и типы полей, типы собственно записей различны, поэтому такая операция присваивания t_StudentRecord1 := t_StudentRecord2 неверна.

Однако типы полей совпадают, поэтому следующие операции верны:

t_StudentRecord1. StudentID := t_StudentRecord2 . StudentID;

t_StudentRecord1. FirstName := t_StudentRecord2 . FirstName;

* Управляющие структуры PL/SQL

Структуры управления являются основой любого языка программирования, поскольку большинство реальных приложений должно уметь обрабатывать множество различных ситуаций. Основную часть структур управления выполнением программы составляют различного рода условные операторы, способные обнаружить существование той или иной ситуации, а за тем инициировать выполнение необходимых действий.

Управление ходом выполнения программы. Конкретная последовательность выполнения различных операторов программы определяется значениями ее переменных и содержанием информации, читаемой из базы данных и записываемой в нее.

Три типа условного оператора IF. При написании компьютерных программ неоднократно возникают ситуации, когда требуется проверить выполнение того или иного условия и в случае, если оно выполняется (имеет место логическое значение TRUE), осуществить одни действия, а при невыполнении условия (логическое значение FALSE) - другие. В языке PL/SQL предусмотрено три типа условного оператора if:

- ✔ **Конструкция IF-THEN**. Эта форма условного оператора предназначена для проверки простых условий. Если условие верно (TRUE), то выполняется одна или несколько строк программы, указанных в теле оператора. Если условие не выполняется (FALSE), то управление передается на следующий оператор.
- ✔ **Конструкция IF-THEN-ELSE.** Эта форма условного оператора аналогична предыдущей,но при невыполнении условия (FALSE) управление передается на один или несколько операторов, указанных после ELSE.
- ✔ **Конструкция IF-THEN-ELSIF.** Этот формат является альтернативой использованию вложенных операторов IF-THEN-ELSE.

Пример

```
DECLARE
        V_num1    NUMBER;
        V_num2    NUMBER;
        V_REZ    VARCHAR2(7);
BEGIN
        …..
            IF     V   _num1 < V_num2
            THEN
                    V_REZ := ‘YES’;
             ELSE
                    V_REZ := ‘NO’;
             END   IF;
END;
```

* Пример

```
IF quantity > 15
      THEN …;     -- скидка 15%
   ELSIF quantity > 10
      THEN …;    -- скидка 10%
   ELSIF quantity > 5
      THEN …;    -- скидка 5%
   ELSE  …;        -- нет скидки
  ENDIF
```

* Циклы

Четыре вида операторов цикла. Циклы позволяют организовать многократное выполнение одного и того же участка программы до полного завершения обработки.

Конструкция LOOP-EXIT-END LOOP

Пример:

```
DECLARE
      V_Counter  INTEGER := 1;
BEGIN
      LOOP
      INSERT INTO   temp_table  VALUES
      (V_Counter, 'LOOP index');
      V_Counter := V_Counter  + 1;
            IF    V_Counter  > 50   THEN
                  EXIT;
                  END   IF;
      END    LOOP;
END;
```

Конструкция LOOP-EXIT WHEN-END LOOP

Оператор EXIT WHEN условие эквивалентен оператору : IF условие THEN EXIT; END IF;

Пример:

```
DECLARE
    V_Counter  INTEGER := 1;
BEGIN
    LOOP
        INSERT INTO   temp_table  VALUES
        (V_Counter, 'LOOP index');
        V_Counter := V_Counter  + 1;
    EXIT   WHEN V_Counter  > 50
    END    LOOP;
END;
```

Конструкция WHILE-LOOP-END LOOP

Пример:

```
DECLARE
        V_Counter  INTEGER
    BEGIN
        WHILE   V_Counter <= 50  LOOP
                INSERT INTO   temp_table  VALUES
          (V_Counter, 'LOOP index');
                V_Counter := V_Counter  + 1;
        END    LOOP;
    END;
```

Конструкция FOR-IN [REVERSE] -LOOP-END LOOP

Пример: BEGIN

FOR V_Counter IN 1..50 LOOP

INSERT INTO temp_table VALUES (V_Counter, 'LOOP index');

END LOOP;

END;

При использовании REVERSE (обратный порядок) индекс цикла будет изменяться от верхней границы до нижней, в следующем примере цикл начнется с 50 и каждый раз будет уменьшаться на 1.

Пример: BEGIN

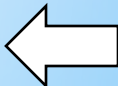
FOR V_Counter IN REVERSE 1..50 LOOP

INSERT INTO temp_table VALUES (V_Counter, 'LOOP index');

END LOOP;

END;

Верхняя и нижняя границы цикла могут быть любыми выражениями, для которых возможно преобразование в числовые значения.



* Присваивание переменным значений базы данных

В зависимости от числа возвращаемых запросом строк используются два метода.

SELECT … INTO … - когда возвращается 1 строка

```
BEGIN
    SELECT id, customerid, orderdate
        INTO currentorder.id, currentorder.customerid,
                currentorder.orderdate
        FROM orders
        WHERE id=453;
```

Если по запросу возвращаются несколько строк, нужно воспользоваться курсором.

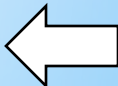
* Курсоры

Курсор - это указатель на контекстную область с помощью которого программа PL/SQL может управлять контекстной областью и ее состоянием во время обработки оператора.

В языке PL/SQL курсоры используются для управления обработкой SQL-операторов select.

Курсоры представляют собой области памяти, специально предназначенные для обработки этих операторов.

В одних случаях курсоры объявляются явно, а других программист предоставляет PL/SQL самому выполнить эту операцию.



* Явно объявляемые курсоры

Явное объявление курсора производится в секции DECLARE, причем указанный в определении SQL-оператор может содержать команды select.

Команды insert, update или delete здесь не допускаются. Явные курсоры используются для обработки тех операторов, которые возвращают более одной строки.

Обработка явных курсоров

Для обработки явного курсора в PL/SQL необходимо выполнить 4 шага:

1. Объявить курсор.
2. Открыть курсор для запроса.
3. Выбрать результаты в переменные PL/SQL.
4. Закрыть курсор.

* Обработка явных курсоров

1) Объявление курсора

При объявлении курсора ему назначается имя и ставится в соответствие некоторый оператор SELECT.

Синтаксис объявления курсора таков:

CURSOR *имя_курсора* IS *оператор_select*

где *имя_курсора* - это имя курсора, *оператор_select* - запрос, который будет обрабатываться.

2) Открытие курсора для запроса

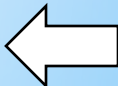
Синтаксис открытия курсора таков:

OPEN *имя_курсора;*

где *имя_курсора* - предварительно объявленный курсор.

Когда курсор открывается, происходит следующее:

- ✔ Анализируются значения переменных привязки.
- ✔ На основе значений переменных привязки определяется активный набор.
- ✔ Указатель активного набора устанавливается на первую строку.



3) Выбор результатов в переменные PL/SQL

Производится считывание строк из курсора. Частью оператора FETCH является список INTO. Оператор FETCH имеет две формы:

FETCH *имя_курсора* INTO *список_переменных*;

или

FETCH *имя_курсора* INTO *запись_ PL/SQL;*

где *имя_курсора* - обозначает предварительно объявленный и открытый курсор,

список_переменных - представляет собой список предварительно объявленных переменных PL/SQL, разделенных запятыми,

запись_ PL/SQL - предварительно объявленная запись PL/SQL.

Переменные в конструкции INTO должны иметь тип, совместимый со списком выбора запроса.

4) Закрытие курсора

Когда выбран весь активный набор, курсор следует закрыть.

Это означает, что программа закончила работу с курсором и отведенные для него ресурсы могут быть освобождены.

Синтаксис закрытия курсора таков:

CLOSE *имя_курсора;*

где *имя_курсора* - ранее открытый курсор.

Курсорные атрибуты

В PL/SQL существует 4 атрибута, которые применимы к курсорам:

- ✔**%FOUND** – это логический атрибут. Он возвращает TRUE, если при предшествующем считывании была выбрана строка, FALSE – если строка выбрана не была.
- ✔**%NOTFOUND** ведет себя противоположно %FOUND. Этот атрибут часто используется в качестве условия выхода из цикла выборки.
- ✔**%ISOPEN** – этот логический атрибут используется для определения, открыт или нет соответствующий курсор. Если открыт, то возвращает TRUE.
- ✔**%ROWCOUNT** – этот числовой атрибут возвращает число строк, считанных курсором на данный момент.

Неявно объявляемые курсоры

Оператор select указывается в теле блока, и PL/SQL берет на себя всю заботу об определении курсора, выполняя соответствующие действия неявно. При этом программисту не требуется вносить в секцию DECLARE никаких дополнительных объявлений.

Обработка неявных курсоров

Каждый оператор select выполняется в пределах контекстной области и поэтому имеет курсор, указывающий на конкретную контекстную область. Такой курсор называется SQL-курсором. В отличии от явных курсоров SQL-курсор не открывается и не закрывается программой. PL/SQL неявно открывает SQL-курсор, обрабатывает SQL- оператор и в последствии закрывает этот курсор, поэтому команды OPEN, FETCH, CLOSE не нужны.

Неявные курсоры используются для обработки операторов INSERT, UPDATE, DELETE, а также однострочных операторов SELECT…INTO

* Пример явного(explicit) курсора

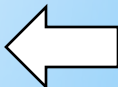
```
DECLARE
        /*Выходные переменные для хранения результатов запроса */
              v_StudentID  students. Id%TYPE;
              v_FirstName students. fitst_name%TYPE;
              v_ LastName students. last_name%TYPE;
        /*  Переменная привязки, используемая в запросе*/
             v_Major    students.major%TYPE := 'Computer Science';
        /*  Создание курсора*/
        CURSOR c_Students IS
              SELECT  id, first_name, last_name
              FROM students
                    WHERE   major =  v_Major;
BEGIN
             /*Обозначим строки активного набора*/
                 OPEN c_Students
                LOOP
            /*Выберем каждую строку  активного набора в переменные PL/SQL*/
                    FETCH c_Students  INTO v_StudentID, v_FirstName, v_ LastName;
            /*  Если строки, которые нужно выбрать, закончились, выйдем из цикла*/
                     EXIT  WHEN c_Students%NOTFOUND;
                 END  LOOP;
           /*  Освободим ресурсы, используемые запросом*/
                CLOSE   c_Students ;
END;
```

* Пример неявного(implicit) курсора

```
BEGIN
        UPDATE    rooms
                  SET   number_seats = 100
                   WHERE   room_id = 999;
/*   Если предыдущий оператор  UPDATE не выбирает ни одной строки, то введем новую строку
   в таблицу rooms*/
        IF    SQL%NOTFOUND THEN
              INSERT     INTO   rooms (room_id, number_seats)
              VALUES (999,  100);
        END IF;
END;
```

Эту же задачу можно выполнить при помощи атрибута SQL%ROWCOUNT:

```
BEGIN
        UPDATE    rooms
                  SET   number_seats = 100
                   WHERE   room_id = 999;
/*   Если предыдущий оператор  UPDATE не выбирает ни одной строки, то введем новую строку
   в таблицу rooms*/
                IF    SQL%ROWCOUNT    THEN
                 INSERT     INTO   rooms (room_id, number_seats)
                 VALUES (999,  100);
                END IF;
END;
```



* Пример

CURSOR *ordercursor* IS *select id, customerid, orderdate from orders;*

DECLARE

CURSOR *ordercursor (ordernumber NUMBER)* IS

SELECT id, customerid, orderdate FROM orders

WHERE id > ordernumber;

BEGIN

OPEN *ordercursor (3)*

В данном примере возвращаемый набор *ordercursor* включает строки таблицы *orders*, для которых идентификатор id > 3

Оператор GOTO

GOTO <метка> - оператор безусловного перехода

Обработка ошибок (блок EXCEPTION)

PL/SQL имеет встроенные исключительные ситуации

no_data_found, too_many_rows, invalid_number, …

EXCEPTION

When *no_data_found* **then**

…

When *too_many_rows* **then**

…

END

Процедура **RAISE_APPLICATION_ERROR (ERRNUM, ERRMES)**

ERRNUM – пользователь задает номер ошибки от -20000 до -20999

*

* Процедуры

Создание процедуры

Синтаксис оператора CREATE OR REPLACE PROCEDURE таков:

CREATE [OR REPLACE] PROCEDURE *имя_процедуры*

[(*аргумент* [{IN | OUT |IN OUT}] *тип*,

. . .

а*ргумент* [{IN | OUT |IN OUT}] *тип*}] {IS | AS}

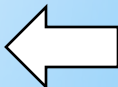
тело_процедуры

где ***имя_процедуры*** - это имя создаваемой процедуры,

аргумент - имя параметра процедуры,

тип - это тип соответствующего параметра,

тело_ процедуры - блок PL/SQL, в котором содержится текст процедуры.



Тело процедуры

Тело (body) процедуры - это блок PL/SQL, содержащий раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. В описании процедуры ключевое слово DECLARE отсутствует.

Как и в анонимных блоках обязательным является только выполняемый раздел. Таким образом, структура процедуры такова:

```
CREATE OR REPLACE PROCEDURE имя_процедуры AS
        /* Раздел объявлений. */
BEGIN
        /* Выполняемый раздел. */
EXCEPTION
        /* Раздел исключительных ситуаций. */
END [имя_процедуры];
```

Для изменения текста процедуры необходимо удалить и повторно создать ее. Во время разработки процедур эта операция выполняется достаточно часто, поэтому ключевые слова OR REPLACE (или заменить) позволяют выполнить такую операцию за один раз.

Ограничения на формальные параметры

При вызове процедуры ей передаются значения фактических параметров, и внутри процедуры к этим значениям обращаются с помощью формальных параметров. При этом передаются не только значения, но и ограничения, наложенные на переменные. Описывая процедуры, запрещается ограничивать длину параметров типа CHAR и VARCHAR2, а также точность и/или масштаб параметров типа NUMBER.

%TYPE и параметры процедур

Единственным способом наложения ограничения на формальные параметры является использование атрибута %TYPE. Если формальный параметр объявлен при помощи %TYPE, а базовый тип ограничен, это ограничение распространяется не на фактический параметр, а на формальный.

Значения параметров по умолчанию

Как и переменные, формальные параметры процедуры или функции могут иметь значения по умолчанию. В таком случае параметр можно не передавать из вызывающей среды. Если же параметр передается, вместо значения по умолчанию берется фактический параметр. Значение по умолчанию для параметра указывается следующим образом:

имя_параметра **[*вид*]** ***тип_параметра***

{:= | DEFAULT} ***исходное _значение***

где *имя _параметра* - это имя формального параметра, *вид* - вид параметра ((IN, OUT или IN OUT), *тип _параметра* - тип параметра, *исходное _значение* - значение, присваиваемое формальному параметру по умолчанию.

Можно применять или символы : = , или ключевое слово DEFAULT.

Удаление процедур

Процедуры и функции, как и таблицы, могут быть удалены.
Синтаксис удаления процедуры выглядит следующим образом:

DROP PROCEDURE ***имя _процедуры;***

Хранимые процедуры

Хранимые процедуры - приложение, объединяющее запросы и процедурную логику и хранящееся в базе данных.
Хранимые процедуры позволяют содержать вместе с БД достаточно сложные программы, выполняющие большой объем работы без передачи данных по сети и взаимодействия с клиентом.

* Пример процедуры

```
CREATE PROCEDURE deletecustomer (custid IN INTEGER) AS
    last VARCHAR2(50);
    first VARCHAR2(50);
BEGIN
   SELECT lastname, firstname INTO last, first
        FROM customer WHERE id=custid;
   INSERT INTO customerhistory VALUES (custid, last, first)
  DELETE FROM customer WHERE id=custid;
EXCEPTION
     WHEN no_data_found THEN
RAISE_APPLICATION_ERROR (-20123, ‘invalid Customer ID’)
END deletecustomer;
```

* Функции

Создание функций

Функции очень похожи на процедуры. Как те, так и другие принимают аргументы, которые могут иметь любой вид.

Функции и процедуры - это различные формы блоков PL/SQL, в состав каждого из них могут входить раздел объявлений, выполняемый раздел и раздел исключительных ситуаций. Как функции, так и процедуры можно хранить в базе данных или описывать в блоке. Однако вызов процедуры сам по себе является оператором PL/SQL, в то время как вызов функции - это часть некоторого выражения.

Как и для процедур, список аргументов необязателен. В этом случае ни при описании функции, ни при ее вызове круглые скобки указывать не нужно. Однако тип, возвращаемый функцией, необходим, так как вызов функции является частью некоторого выражения. Тип функции используется для определения типа выражения, содержащего вызов этой функции.

Описание функций

Синтаксис для создания хранимой функции очень похож на синтаксис для создания процедуры:

CREATE [OR REPLACE] FUNCTION *имя_функции*

[(*аргумент* [{IN | OUT |IN OUT}] *тип*,

. . .

***аргумент*[{IN | OUT |IN OUT}] *тип*)]**

RETURN *возвращаемый _тип* {IS | AS} *тело_функции*

где *имя_функции* - это имя функции;

аргумент и *тип* аналогичны аргументу и типу, указываемым при создании процедуры;

возвращаемый_тип - это тип значения, возвращаемого функцией;

тело_функции - блок PL/SQL, содержащий программный текст данной функции.

Оператор RETURN

Внутри тела функции оператор RETURN применяется для возврата управления программой и результата выполнения функции в вызывающую среду. Общий синтаксис оператора RETURN выглядит следующим образом:

RETURN *выражение*,

где *выражение* - это возвращаемое значение. Значение выражения преобразуется к типу, указанному в команде RETURN при описании функции, если это значение уже не имеет данный тип. При выполнении оператора RETURN управление программой сразу же возвращается в вызывающую среду.

В функции может быть несколько операторов RETURN, хотя выполняться будет только один из них. Завершение функции без оператора RETURN является ошибкой.

Свойства функций

Многие из свойств функций аналогичны свойствам процедур:

- ✔ Функции могут возвращать более одного значения при помощи параметра вида OUT.
- ✔ Программный код функции состоит из раздела объявлений, выполняемого раздела и раздела исключительных ситуаций.
- ✔ Функции могут использовать значения по умолчанию.
- ✔ Функции можно вызывать, используя позиционное или именное представление.

Когда применять функцию, а когда процедуру зависит от того, сколько значений должна возвращать данная подпрограмма и как будут использоваться эти значения. Обычно принято следующее правило: если возвращается более одного значения, нужно использовать процедуру, а если ровно одно, то функцию.

Удаление функций

Процедуры и функции, как и таблицы, могут быть удалены. При выполнении этой операции процедура или функция удаляется из словаря данных. Синтаксис удаления функции выглядит следующим образом:

DROP FUNCTION *имя_функции*.

* Пример функции

```
CREATE OR REPLACE FUNCTION AlmostFull(
   p_Department classes.department%TYPE,
   p_Course   classes.course%TYPE)
RETURN BOOLEAN IS
   V_CurrentStudents  NUMBER;
   V_MaxStudents   NUMBER;
   V_ReturnValue   BOOLEAN;
   V_FullPercent     CONSTANT   NUMBER := 90;
BEGIN       /*Узнаем текущее и максимальное число студентов в указанной группе*/
      SELECT  current_students, max_students
            INTO  V_CurrentStudents, V_MaxStudents
            FROM  classes
            WHERE  department = p_Department   AND   course = p_Course;
   /*Если процент заполнения группы более заданного  в V_FullPercent */

   IF   (V_CurrentsStudents / V_MaxStudents * 100) > V_FullPercent
   THEN
      V_ReturnValue := TRUE;
   ELSE
      V_ReturnValue := FALSE;
   END    IF;
   RETURN  V_ReturnValue;
END  AlmostFull;
```

* Пример функции

```
CREATE FUNCTION findcustid (last IN VARCHAR2, first IN VARCHAR2)
RETURN INTEGER  AS
  custid INTEGER;
BEGIN
    SELECT id INTO custid   FROM customer
      WHERE lastname=last AND firstname=first;
    RETURN custid;
EXCEPTION
     WHEN no_data_found THEN
          RAISE_APPLICATION_ERROR (-20101, ‘invalid Customer ID’)
END findcustid;
```

* Агрегирующие функции

Групповые функции обрабатывают по несколько строк, но возвращают один результат. Эти функции можно применять только в списках выбора запросов и в конструкции GROUP BY.

В большинстве этих функций допускается использование квалификаторов (уточнителей) аргументов: DISTINCT (отличные от других) и ALL (все). Если указывается DISTINCT, рассматриваются только те значения, возвращаемые запросом, которые отличны от других. Когда используется квалификатор ALL, функция рассматривает все значения, возвращаемые запросом. Если не указано другое условие, ALL принимается как параметр, заданный по умолчанию.

MAX

Синтаксис	MAX([DISTINCT / ALL]
Назначение	Возвращает максимальное значение для пункта списка выбора
Область применения	Только списки выбора запросов и конструкции GROUP BY

COUNT

Синтаксис	COUNT([* / DISTINCT / ALL]
Назначение	Возвращает число строк в запросе. Если указана *, возвращается общее число строк. Если указан пункт сипска выбора, то подсчитываются не-NULL значения
Область применения	Только списки выбора запросов и конструкции GROUP BY

AVG

Синтаксис	AVG([DISTINCT / ALL]
Назначение	Возвращает среднее для значения столбца
Область применения	Только списки выбора запросов и конструкции GROUP BY

MIN

Синтаксис	MIN([DISTINCT / ALL]
Назначение	Возвращает минимальное значение для пункта списка выбора
Область применения	Только списки выбора запросов и конструкции GROUP BY

SUM

Синтаксис	SUM([DISTINCT / ALL]
Назначение	Возвращает сумму значений для пункта списка выбора
Область применения	Только списки выбора запросов и конструкции GROUP BY

* Модули (Пакеты)

Модуль - это конструкция PL/SQL, позволяющая хранить связанные объекты в одном месте.

Модуль состоит из двух различных частей: описания и тела, каждая из которых хранится по отдельности в словаре данных.

В отличие от процедур и функций, которые содержатся локально в блоке или хранятся в базе данных, модули могут быть только хранимыми и никогда локальными.

Модули позволяют объединять связанные объекты, а также используют менее ограничений, определяемых зависимостями. Кроме того, они имеют ряд свойств, повышающих производительность системы.

В сущности, модуль представляет собой именованный раздел объявлений. Все входящее в состав раздела объявлений блока, может входить и в модуль: процедуры, функции, курсоры, типы и переменные. Размещение их в модуле позволяет ссылаться на них из других блоков PL/SQL, поэтому в модулях можно описывать глобальные переменные для PL/SQL.

*Описание модуля (Пакета)

CREATE [OR REPLACE] PACKAGE *имя_модуля* {IS |AS}

описание_процедуры
описание_функции
объявление_переменной
определение_типа
объявление_исключительной_ситуации
объявление_курсора
END [*имя_модуля*];

где *имя_модуля* - это имя модуля. Элементы модуля (описания процедур и функции, переменные и т.д.) аналогичны указанным в разделе объявлении анонимного блока.

Для заголовка модуля верны те же синтаксические правила, установленные для раздела объявлений, за исключением объявлений процедуры и функции.

Перечислим эти правила:

1.Элементы модуля могут указываться в любом порядке. Однако, как и в разделе объявлений, объект должен быть объявлен до того, как на него будут произведены ссылки. Например, если частью условия WHERE курсора является некоторая переменная, то она должна быть объявлена до объявления курсора.

2.Присутствие элементов всех видов совсем не обязательно.

3.Объявления всех процедур и функций должны быть предварительными. В этом отличие модуля от раздела объявлений блока, где могут находиться как предварительные объявления, так и реальный текст процедур и функций.

Тело модуля

Тело модуля (package body) - это объект словаря данных, хранящийся отдельно от заголовка модуля. Тело модуля нельзя успешно скомпилировать без успешной компиляции заголовка.

В теле содержится текст подпрограмм, предварительно объявленных в заголовке модуля.

Тело модуля не является обязательной его частью. Если в заголовке не указаны какие-либо процедуры или функции (а только переменные, курсоры, типы и т.д.), тело можно не создавать.

Любое предварительное объявление в заголовке модуля должно быть раскрыто в его теле. Описание процедуры или функции должно быть таким же и включать в свой состав имя подпрограммы, имена ее параметров и вид каждого параметра.

Модули и области действия

Любой объект, объявленный в заголовке модуля, находится в области действия и видим вне границ этого модуля. Для обращения к объекту нужно указать имя модуля при ссылке на этот объект.
При этом вызов процедуры аналогичен вызову процедуры, не включенной в модуль. Единственное отличие такого вызова - присутствие перед именем процедуры имени модуля. Для модульных процедур могут задаваться параметры по умолчанию, и вызывать такие процедуры можно при помощи как позиционного, так и именного представления, то есть точно так же, как и обычные хранимые процедуры

Кроме того, в модуле можно применять типы данных, определяемые пользователями.

Инициализация модуля

При вызове первый раз модуль конкретизируется (instrantiated). Это значит, что модуль считывается с диска в память, а затем запускается p-код. В этот момент для всех переменных, описанных в модуле, выделяется память. У каждого сеанса будет собственная копия модульных переменных: это гарантирует, что два сеанса, выполняющие подпрограммы одного и того же модуля, будут использовать различные области памяти.

Во многих случаях код инициализации нужно запускать на выполнение при первой конкретизации модуля. Это можно сделать, если к телу модуля добавить раздел инициализации, разместив его после всех объектов:

CREATE OR REPLACE PACKAGE BODY *имя_модуля* {IS | AS}

. . .

BEGIN

код_инициализаци;

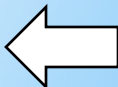
END [*имя_модуля*];

где *имя_модуля* – имя модуля,

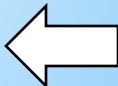
***код_инициализации* – запускаемый код.**

Пример модуля генерации случайных чисел

```
CREATE OR REPLACE PACKAGE Random AS
PROCEDURE ChangeSeed (p_NewSeed IN NUMBER);
FUNCION Rand RETURN NUMBER;
PROCEDURE GetRand (p_RandomNumber OUT NUMBER);
FUNCION RandMax (p_MaxVal IN NUMBER) RETURN
  NUMBER;
PROCEDURE GetRandMax (p_RandomNumber OUT NUMBER,
  p_MaxVal IN NUMBER);
END Random;

CREATE OR REPLACE PACKAGE BODY Random AS
        v_Multiplier CONSTANT NUMBER := 22695477;
        v_Increment CONSTANT NUMBER := 1;
        v_Seed NUMBER :=1;
```



```
PROCEDURE ChangeSeed(p_NewSeed IN NUMBER) IS
BEGIN
  v_Seed := p_NewSeed;
END ChangeSeed;

FUNCTION Rand RETURN NUMBER IS  /*Возвращает
  случайное число в диапазоне от 1 до 32767*/
BEGIN
  v_Seed :=MOD(v_Multiplier * v_Seed + v_Increment, (2 ** 32));
  RETURN  BITAND (v_Seed/(2 ** 16), 32767);
END Rand;

PROCEDURE GetRand(p_RandomNumber OUT NUMBER) IS
  /*Аналогична функции Rand, но с процедурным интерфейсом*/
BEGIN
  p_RandomNumber := Rand;
END GetRand;
```



```
CREATE OR REPLACE PACKAGE ClassPackege AS
PROCEDURE AddStudent(p_StudentId IN Students. Id %TYPE,
p_Department IN classes.departmen%TYPE, p_Courses IN classes.course%TYPE
);
PROCEDURE RemoveStudent(p_StudentId IN Students.ID%TYPE,
p_Department IN classes.departmen%TYPE,
p_Courses IN classes.course%TYPE );
E_studentNotRegistered EXCEPTION;

TYPE t_StudentIDTable IS TABLE OF Students.Id %TYPE
INDEX BY BINARY_INTEGER;

PROCEDURE ClassList(p_Department IN classes.departmen%TYPE,
p_Courses IN classes.course%TYPE , pIDS OUT t_StudentIDTable,
p_NumStudents IN OUT BINARY_INTEGER );
END ClassPackege;
```

В этом модуле содержится описание трех процедур, одного типа и исключительной ситуации.

```
FUNCTION RandMax(p_MaxVal IN NUMBER) RETURN NUMBER  IS
BEGIN
--Возвращает случайное целое число в диапазоне от 1 до p_MaxVal      RETURN MOD (Rand,  p_MaxVal)  +  1;
END  RandMax;


PROCEDURE  GetRandMax(p_RandomNumber   OUT   NUMBER,  p_MaxVal  IN NUMBER)  IS
BEGIN
      p_RandomNumber := RandMax (p_MaxVal);
END  GetRandMax;
BEGIN
/* Инициализация модуля.  Инициализируем  исходное  значение  текущим временем  в секундах */
  ChangeSeed(TO_NUMBER (TO_CHAR(SYSDATE, 'SSSS')));
END  Random;
```

Для получения случайного числа можно просто вызвать Random.Rand. Последовательность случайных чисел зависит от исходного значения – для одного и того же исходного значения генерируются одинаковые последовательности.

* Пример модуля (пакета)

```
CREATE OR REPLACE PACKAGE customermanager IS
PROCEDURE newcustomer (company IN VARCHAR2 DEFAULT null, last IN
   VARCHAR2, first IN VARCHAR2, …);
FUNCTION findcustid (last IN VARCHAR2, first IN VARCHAR2)
 RETURN INTEGER;
   PROCEDURE updatecustomer (custid IN INTEGER, fieldtype IN CHAR,
   newvalue IN VARCHAR2);
PROCEDURE deletecustomer (custid IN INTEGER);
PROCEDURE deletecustomer (last IN VARCHAR2, first IN VARCHAR2);
END customermanager;

CREATE  OR REPLACE PACKAGE BODY customermanager AS
….
END customermanager;
```

* Триггеры

Триггеры так же, как процедуры и функции, являются именованными блоками PL/SQL с разделом объявлений, выполняемым разделом и разделом исключительных ситуаций.

Подобно модулям, триггеры необходимо хранить в базе данных, а не локально в блоке.

Триггер выполняется неявно, всякий раз, когда происходит событие, запускающее этот триггер, причем использование аргументов не допускается.

Акт выполнения триггера называется его активизацией (firing).

Запускается триггер операцией DML (INSERT, UPDATE или DELETE), выполняемой над базой данных.

* Создание триггеров

CREATE [OR REPLACE] TRIGGER *имя_триггера*
{BEFORE | AFTER} активизирующее событие **ON** *ссылка_на_таблицу*
[FOR EACH ROW [WHEN *условие_срабатывания*]] *тело_триггера;*

где *имя_триггера* – имя триггера;

активизирующее событие – момент активации триггера;

ссылка_на_таблицу – таблица для которой создан триггер;

условие_срабатывания – если оно есть, то сначала оно вычисляется, и только если условие это истинно, срабатывает тело триггера;

тело_триггера – программный текст триггера;

BEFORE-триггеры используются для проверки правильности и/или модификации вводимых данных, например перевод в верхний регистр и тп.

AFTER-триггеры выполняют действия с данными уже обработанными командой DML, например протоколирование действий пользователя и др.

Триггеры можно использовать для:

- ✔ Реализации сложных ограничений целостности данных, которые невозможно осуществить через описательные ограничения, устанавливаемые при создании таблицы
- ✔ Слежения за информацией, хранимой в таблице, путем записи вносимых изменений и пользователей, вносящих эти изменения
- ✔ Автоматического оповещения других программ о том, что делать в случае изменения информации, содержащейся в таблице

Типы триггеров

- ❖ Тип триггера определяется тем, какое событие его активизирует: INSERT (ввод), UPDATE (обновление) или DELETE (удаление).
- ❖ Триггеры могут активизироваться до (BEFORE) или после (AFTER) операции, а также для строки или оператора.
- ❖ Триггеры могут активироваться для строки или оператора. Если триггер строковый, то он активизируется один раз для каждой из строк, на которые воздействует оператор, вызывающий срабатывание триггера. Если триггер операторный, то он активизируется один раз до или после оператора. Строковые триггеры содержат условие FOR EACH ROW (для каждой строки оператора) в описании триггера.

Элементы триггера

Обязательными элементами триггера являются его имя, активизирующее событие и тело. Условие WHEN необязательно.

Имена триггеров

Пространство имен триггеров (набор идентификаторов), разрешенных для использования в качестве имен объектов отличается от пространств имен других подпрограмм.

Для процедур, модулей и таблиц применяется одно и то же пространство имен, это значит, что в пределах одной схемы базы данных все объекты, использующие одно и то же пространство имен, должны иметь уникальные имена.

Например, модуль и процедура в одной схеме не могут иметь одинаковых имен, а триггер может иметь то же тмя, что и процедура или модуль. Однако в пределах одной схемы конкретное имя может быть дано только одному триггеру.

Имена триггеров – это идентификаторы базы данных, поэтому подчиняются стандартным правилам для идентификаторов.

Удаление и запрещение триггеров

Триггеры, как и процедуры, и модули, и функции, можно удалять. Синтаксис таков:

DROP TRIGGER *имя_триггера*;

Однако в отличие от процедур и функций, можно не удаляя триггер, запретить (disable) его использование. Когда триггер запрещен, он по-прежнему находится в словаре данных, но никогда не активизируется.

С помощью оператора **ALTER TRIGGER *имя_триггера* {DISABLE| ENABLE};** можно запретить или разрешить любой триггер.

Все триггеры таблицы выключаются/включаются командой:

ALTER TABLE имя_таблицы {DISABLE | ENABLE} ALL TRIGGERS;

Порядок активизации триггера

Триггер активизируется при выполнении оператора DML. Алгоритм выполнения DMLоператора таков:

1. Выполняется операторный триггер BEFORE (при его наличии).

2. Для каждой строки, на которую воздействует оператор:

a. Выполняется строковый триггер BEFORE (при его наличии);

b. Выполняется собственно оператор;

c. Выполняется строковый триггер AFTER (при его наличии);

3. Выполняется операторный триггер AFTER (при его наличии).

Триггер никак не проверяет данные, которые уже были в таблице до того момента, когда он был создан или включен.

Пример содержит все 4 вида триггеров UPDATE (BEFOR и AFTER, операторный и строковый) для таблицы classes.

Создадим числовую последовательность:

```
CREATE SEQUENCE  trigger_seq
  START  WITH   1
  INCREMENT  BY 1;
```

```
CREATE [OR REPLACE] TRIGGER classesBEstatement
    BEFORE UPDATE ON classes
BEGIN
    INSERT INTO temp_table (num_col, char_col)
   VALUES (trigger_seq.NEXTVAL, 'BEFORE ОПЕРАТОРНЫЙ ТРИГГЕР')
END classesBEstatement;

CREATE [OR REPLACE] TRIGGER classesAFstatement
    AFTER UPDATE ON classes
BEGIN
    INSERT INTO temp_table (num_col, char_col)
    VALUES (trigger_seq.NEXTVAL, 'AFTER ОПЕРАТОРНЫЙ ТРИГГЕР')
END classesAFstatement;
```

```
CREATE [OR REPLACE] TRIGGER classesBERow
   BEFORE UPDATE ON classes FOR EACH ROW
BEGIN
   INSERT INTO temp_table (num_col, char_col)
         VALUES (trigger_seq.NEXTVAL, 'BEFORE СТРОКОВЫЙ ТРИГГЕР')
END classesBERowt;

CREATE [OR REPLACE] TRIGGER classesAFRow
AFTER UPDATE ON classes FOR EACH ROW
BEGIN
   INSERT INTO temp_table (num_col, char_col)
         VALUES (trigger_seq.NEXTVAL, 'AFTER СТРОКОВЫЙ ТРИГГЕР')
END classesAFRow;
```

Теперь выполним оператор UPDATE:

```
UPDATE classes
   SET num_credit =4
WHERE department IN ('AA','BB');
```

Этот оператор воздействует на 4 строки. Каждый из операторных триггеров выполняется один раз, а каждый из строковых – 4 раза.

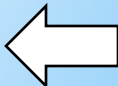
Ограничения, налагаемые на триггеры

Тело триггера является блоком PL/SQL.

Любой оператор, выполнение которого разрешено в блоке PL/SQL, можно выполнить и в теле триггера при условии соблюдения следующих ограничений:

- ✔ В триггере нельзя задавать ни один из операторов управления транзакциями: COMMIT, ROLLBACK или SAVEPOINT.
- ✔ Срабатывание триггера является частью процесса выполнения активизирующего оператора, то есть частью той транзакции, которая охватывает и активизирующий оператор.
- ✔ Когда этот оператор завершается или откатывается, все выполненное триггером также завершается или откатывается.
- ✔ В процедурах и функциях, вызывающихся в теле триггера, также нельзя задавать какие-либо из операторов управления транзакциями.
- ✔ В теле триггера нельзя объявлять переменные с типами LONG и LONG RAW.
- ✔ Кроме того, в псевдозаписях :new и :old (см. ниже) нельзя ссылаться на столбцы типов LONG и LONG RAW таблицы, для которой определен триггер.

Из тела триггера можно обращаться не ко всем таблицам в зависимости от типа триггера и ограничений, накладываемых на таблицы.



Использование :old и :new в строковых триггерах

Строковый триггер срабатывает один раз для каждой строки, обрабатываемой активизирующим оператором.

Внутри триггера можно обращаться к строке, обрабатываемой в данный момент. Для этого служат две псевдозаписи - :old и :new.

Хотя синтаксически они рассматриваются как записи, фактически они записями не являются.

Поэтому их называют псевдозаписями.

Тип обеих псевдозаписей определяется:

***активизирующая_таблица%*ROWTYPE;**

Хотя :old и :new синтаксически рассматриваются в качестве записей типа *активизирующая_таблица%*ROWTYPE, в действительности они записями не являются. Псевдозаписи нельзя присваивать чему-либо целиком, можно только поля псевдозаписей.

:new модифицируется только в строковом триггере BEFORE, а :old никогда не модифицируется, а лишь считывается.

Активизирующий оператор	:old	:new
INSERT	Не определена – во всех полях NULL - значения	Значения, которые будут выведены после выполнения оператора
UPDATE	Исходные значения, содержащиеся в строке перед обновлением данных	Новые значения, которые будут введены после выполнения оператора
DELETE	Исходные значения, содержащиеся в строке перед ее удалением	Не определена – во всех полях NULL - значения

Доступ к значениям столбцов

Доступ к значениям столбцов в триггере осуществляется с помощью корреляционных имен:

:NEW.имя_столбца — новое значение;

:OLD.имя_столбца — старое значение.

В триггере INSERT имеют смысл только новые значения, для DELETE — только старые значения, для UPDATE — оба.

Пример тригерра BEFORE, срабатывающего на операторы INSERT и UPDATE, заполняющий поле ID в таблице Students значением, генерируемым последлвательностью trigger_seq.

```
CREATE [OR REPLACE] TRIGGER GenerateStudentId
   BEFORE INSERT OR UPDATE ON Students FOR EACH ROW
BEGIN
        SELECT student_seq.nextval
              INTO :new.ID
               FROM   dual;
END GenerateStudentId;
```

* Примеры триггеров

1) CREATE TRIGGER deletecustomer
BEFORE DELETE ON customer
FOR EACH ROW
BEGIN
INSERT INTO customerhistory
VALUES (:old.id, :old.lastname, :old.firstname);
END deletecustomer;

2) CREATE TRIGGER timecheck
BEFORE UPDATE OR DELETE ON customer
BEGIN
IF to_char (sysdate, ‘HH24’) NOT BETWEEN 7 or 18 THEN
RAISE_APPLICATION_ERROR (-20101, ‘изменять запись о покупателе в это время не допускается’);
ENDIF
END timecheck;

* Примеры триггеров

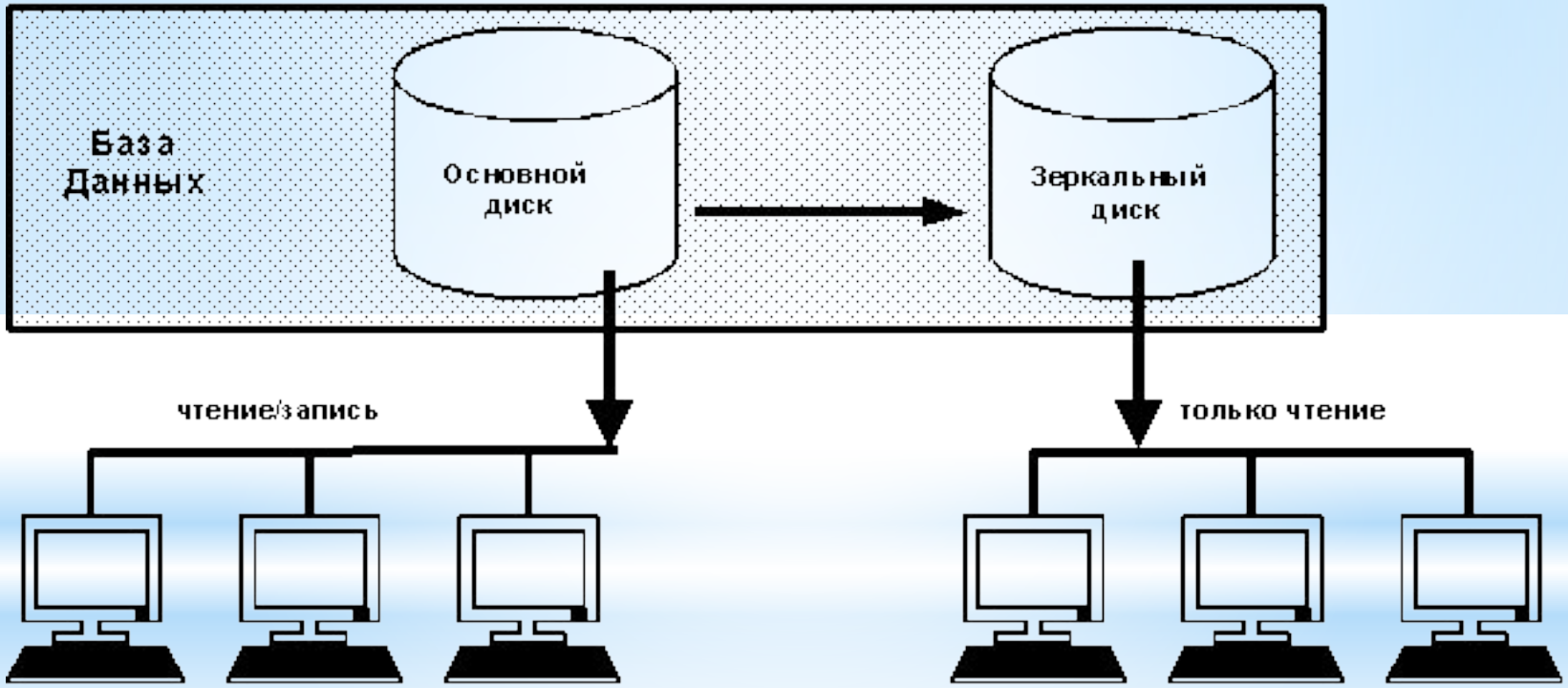
```
3)  CREATE TRIGGER updatestockquantity
      AFTER INSERT OR DELETE OR UPDATE  OF quantity ON item
       FOR EACH ROW
  BEGIN
    IF inserting THEN
       UPDATE stock SET onhand = onhand - :new.quantity
            WHERE id = :new.stockid;
     ELSIF updating THEN
    IF :new.quantity > :old.quantity THEN
  UPDATE stock SET onhand = onhand – (:new.quantity - :old.quantity)
            WHERE id = :new.stockid;
   ELSE
  UPDATE stock SET onhand = onhand + (:old.quantity - :new.quantity)
            WHERE id = :new.stockid;
    ENDIF;
  ELSE     UPDATE stock SET onhand = onhand + :old.quantity
            WHERE id = :new.stockid;
ENDIF;
END updatestockquantity;
```

* Параллельные архитектуры серверов баз данных

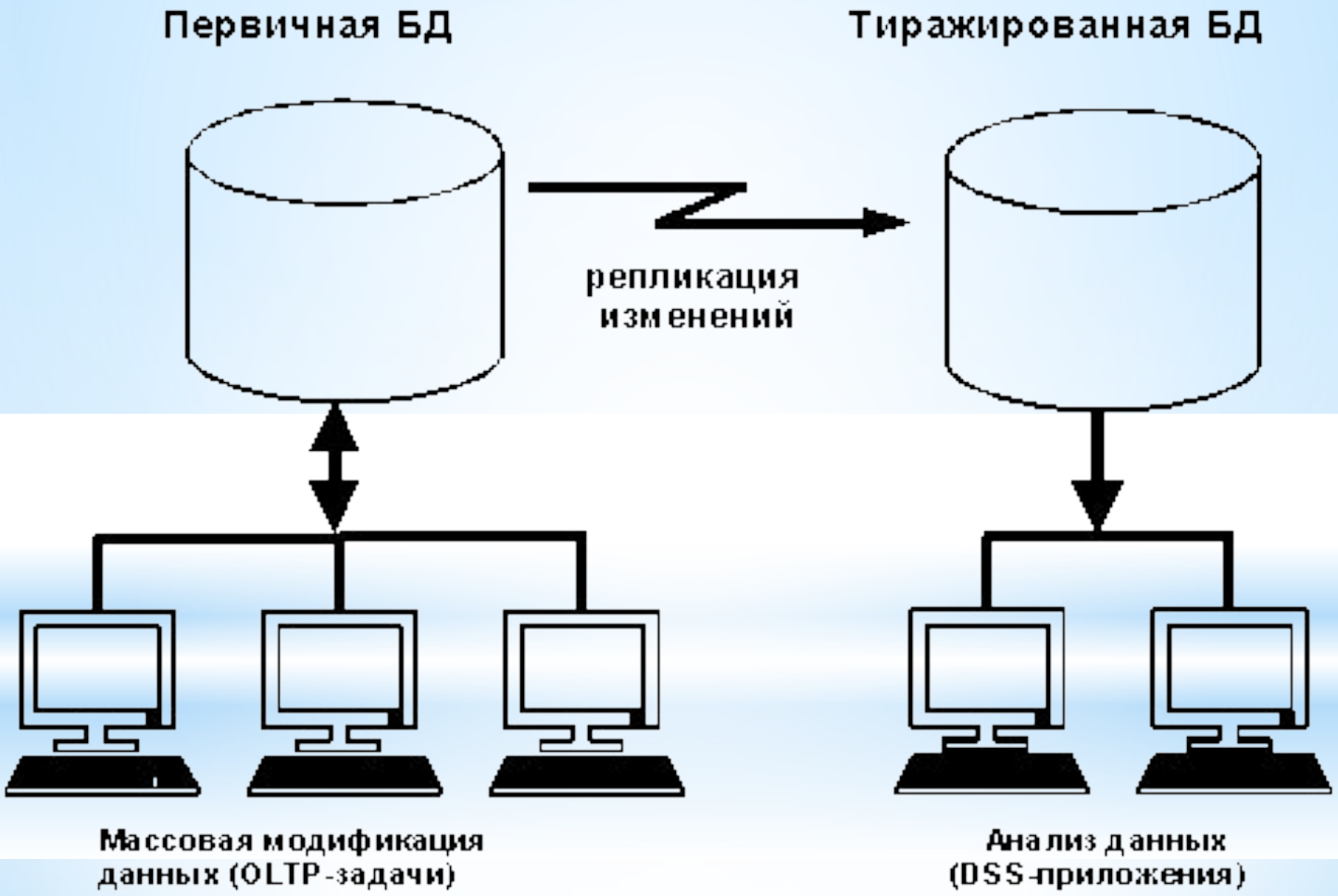
Три основные архитектурные направления:

* Симметричные многопроцессорные системы (SMP) - форма сильносвязанных многопроцессорных систем, разделяющих единую оперативную память и дисковую подсистему;
* Слабосвязанные многопроцессорные системы (кластеры) - совокупность компьютеров, объединенных в единую систему быстродействующей сетью и имеющих общую дисковую подсистему;
* Системы с массовым параллелизмом (MPP) - системы с сотнями и даже тысячами процессоров, имеющие многоуровневую структуру оперативной памяти

* Зеркалирование (software mirroring)



* Тиражирование (replication) данных



* Распределенные системы баз данных

Ядром системы управления распределенными информационными ресурсами являются распределенная база данных и система управления распределенной базой данных.

Распределенная база данных – это совокупность логически взаимосвязанных баз данных, распределенных в компьютерной сети.

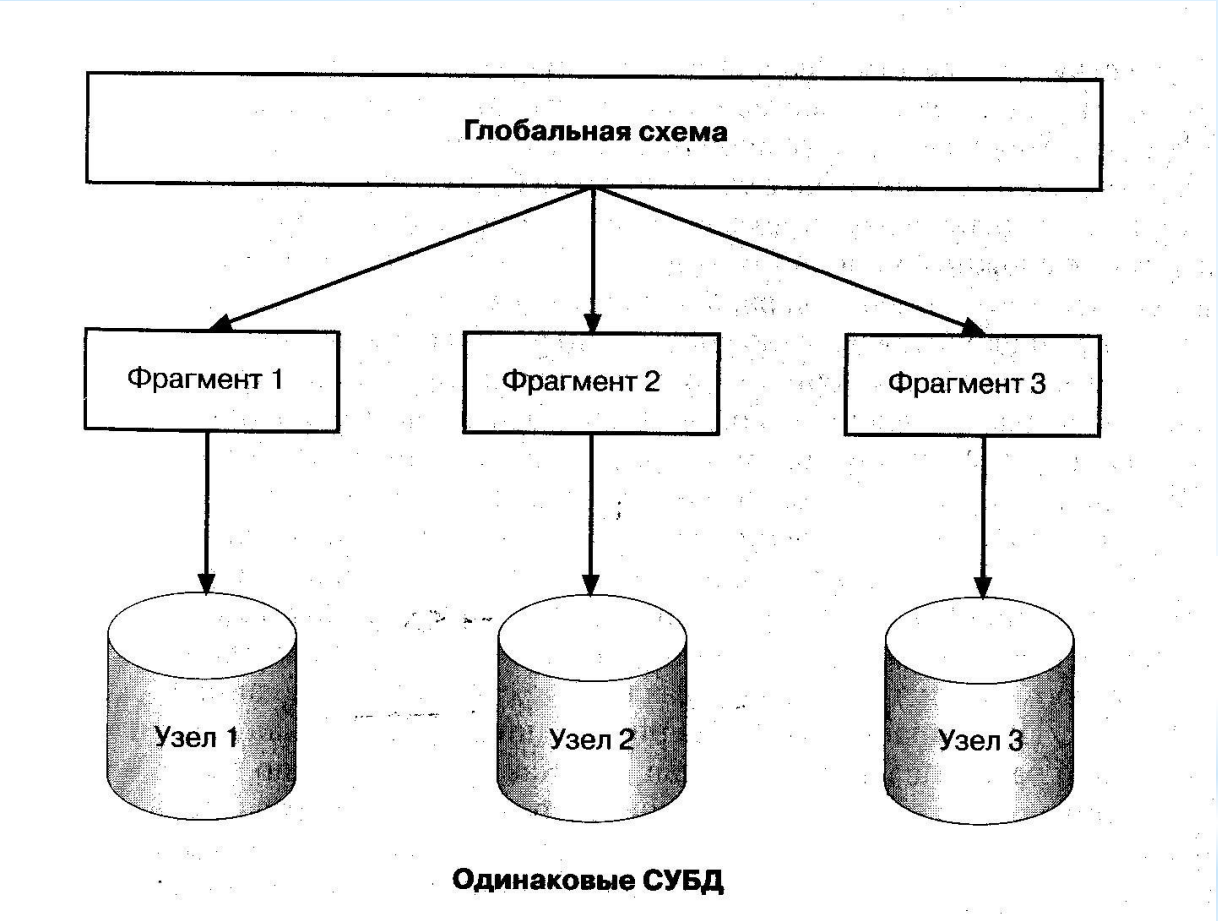
Система управления распределенной базой данных – программная система, которая обеспечивает управление распределенной базой данных и прозрачность ее распределенности для пользователей.

Распределение производится путем фрагментации или тиражирования

* Правила К. Дейта для распределенных баз данных

1. Локальная автономность
2. Никакой конкретный сервис не должен возлагаться на какой-либо выделенный центральный узел.
3. Непрерывность функционирования.
4. Независимость от месторасположения.
5. Независимость от фрагментации.
6. Независимость от тиражирования.
7. Распределенная обработка запросов
8. Управление распределенными транзакциями.
9. Независимость от оборудования.
10. Независимость от операционных систем.
11. Независимость от сети.

* Модели распределенных баз данных



Однородные системы, если СУБД –одинаковые, иначе – неоднородные системы

* Фрагментация и тиражирование

Методы проектирования распределенных баз данных «сверху вниз» и «снизу вверх»

Проектирование «сверху вниз» аналогично проектированию централизованных баз данных:

- создание концептуальной модели базы данных;
- отображение ее в логическую модель данных;
- создание и настройка специфических для СУБД структур.

Однако при проектировании распределенной БД предполагается, что объекты не будут сосредоточены в одном месте, а распределятся по нескольким вычислительным системам.

Распределение проводится путем фрагментации и тиражирования.

Фрагментация означает декомпозицию объектов базы данных (например, таблицы) на несколько частей, которые размещаются на разных системах.

Существуют горизонтальная и вертикальная фрагментация (по строкам или по столбцам).

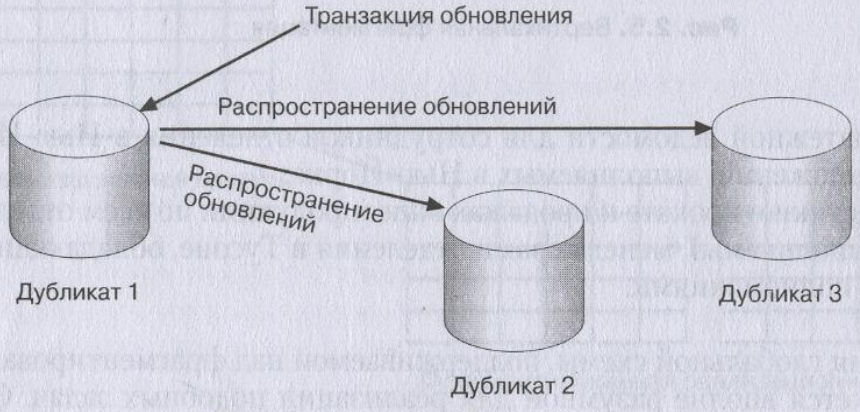
В любом случае поддерживается глобальная схема, позволяющая воссоздать из фрагментов логически централизованную таблицу или другую структуру.

Тиражирование (или репликация) – создание дубликатов данных.

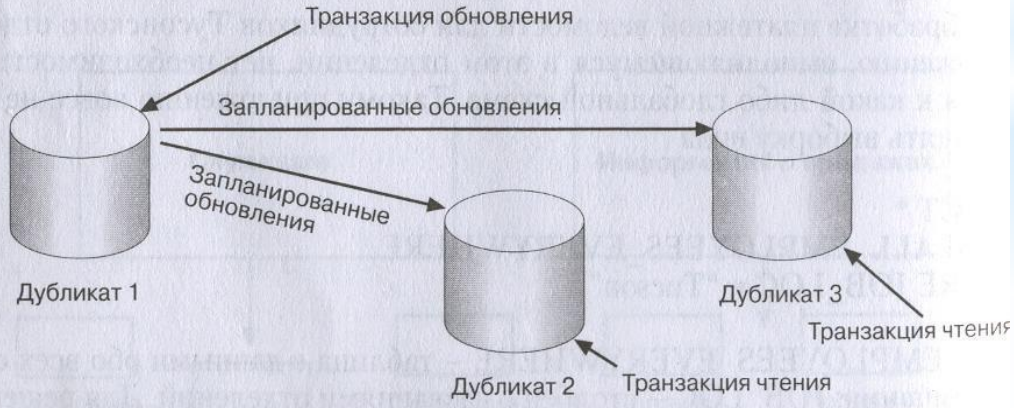
Дубликаты (репликаты) – это множество различных копий некоторого объекта базы данных, для которых в соответствии с определенными правилами поддерживается синхронизация с главной копией.

* Модели тиражирования данных

Одновременное обновление (с управлением параллелизмом)



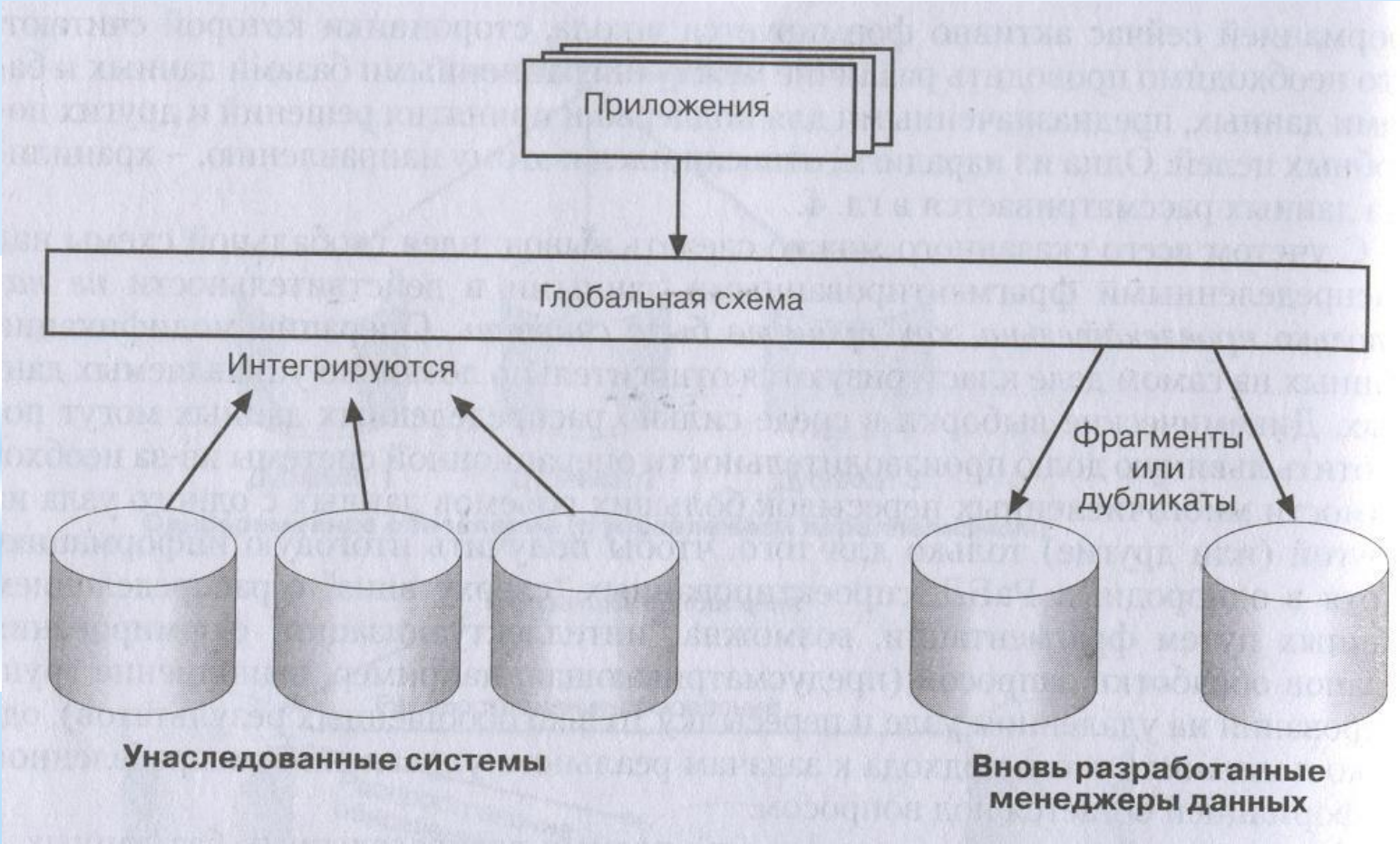
Распространенные обновления



Запланированная синхронизация дубликатов только для чтения

* Интеграция распределенных баз данных «снизу-вверх»

Проектирование распределенных баз данных «снизу вверх» - объединение схем уже существующих БД, чтобы предоставить как новым, так и прежним приложениям доступ и к новым и к старым ресурсам данных (система мультибаз данных).



Доступ к базам данных

Системы прозрачного доступа к БД представляют собой популярное решение. В простых двухзвенных моделях клиент-сервер, где несколько баз данных обслуживают ограниченное число пользователей настольных ПК, в роли встроенного MiddleWare (MW) доступа к данным могут выступать обычные ODBC-драйверы. Необходимость в более сложных решениях возникает в больших, разнородных многозвенных системах, где множество приложений в параллельном режиме осуществляет доступ к разнообразным источникам данных, включая СУБД и хранилища данных от различных поставщиков. В таких системах между клиентами и серверами баз данных размещается промежуточное звено – SQL-шлюз, который представляет собой набор общих API, позволяющих разработчику строить унифицированные запросы к разнородным данным (в формате SQL или с помощью ODBC-интерфейса). SQL-шлюз выполняет синтаксический разбор такого запроса, анализирует и оптимизирует его и выполняет преобразование в SQL-диалект нужной СУБД. MW этого типа реализует синхронный механизм связи, когда выполнение приложения, сделавшего запрос, блокируется до момента получения данных. Надо заметить, что синхронные принципы взаимодействия в распределенной среде, как правило, порождают проблемы масштабируемости системы.

Доступ к базам данных

- Использование MW доступа к БД широко применяется в корпоративных системах поддержки принятия решений (DSS), которые собирают и анализируют данные из множества разнородных источников и не требуют управления оперативными транзакциями.
- Рынок средств прозрачного доступа к базам данных практически не стандартизован – поставщики обычно создают свои частные решения и не обременены проблемами совместимости. Это можно объяснить тем, что приложение, использующее данный тип MW, извлекает информацию непосредственно из статического источника (хранилища данных), а не обращается за ней к другому прикладному модулю, возможно, от другого поставщика.

* Архитектура ODBC(OPEN DATABASE CONNECTIVITY)

* SQL - приложение
  Администратор ODBC
  Драйверы ODBC для различных СУБД
  локальные или удаленные БД

* Основная идея:

  - все операции с базой данных идут через специальный программный слой, не зависящий от СУБД;

  - конфигурация ODBC для каждого источника данных (alias) определяет его драйвер и местоположение;

  - при изменении драйвера или местоположения необходимо изменить эти параметры в конфигурации

* Существует 4 важных этапа (шага) процедуры запроса данных через ODBC API.
* *Шаг 1* - установление соединения. Первый шаг состоит в размещении указателей (handle) среды ODBC, которые выделяют оперативную память под ODBC драйверы и библиотеки. Затем происходит выделение памяти для указателей соединения, и соединение устанавливается.
* *Шаг 2* - выполнение оператора SQL. Выделяется указатель оператора, локальные переменные связываются со столбцами в SQL-выражении (это необязательное действие), и выражение представляется главному ODBC-драйверу для обработки.
* *Шаг 3* - извлечение данных. Перед извлечением данных возвращается информация о результирующем наборе, в частности, число столбцов в наборе. Исходя из этого числа, результирующий набор помещается в буфер записей, выполняется цикл его просмотра и содержимое каждого столбца помещается в соответствующую локальную переменную.
* *Шаг 4* - освобождение ресурсов.
* Технология ODBC разрабатывалась как общий, независимый от источников данных, способ доступа к данным. Применение технологии обеспечивает переносимость приложений в среду различных баз данных без переработки самих приложений. Технология ODBC уже стала промышленным стандартом, ее поддерживают практически все производители СУБД и средств разработки.

* Недостатки реляционных СУБД

- Слабое представление сущностей реального мира
- Семантическая перегрузка
- Слабая поддержка ограничений целостности и корпоративных ограничений
- Однородная структура данных
- Ограниченный набор операций
- Трудности организации рекурсивных запросов
- Проблема рассогласования
- Другие проблемы РСУБД, связанные с параллельностью, изменениями схемы и слабыми средствами доступа

* Манифест систем объектно-ориентированных баз данных

Обязательные свойства: золотые правила

Система объектно-ориентированных баз данных должна удовлетворять двум критериям: она должна быть СУБД и при этом являться объектно-ориентированной системой, т.е. в максимально возможной степени находиться на уровне современных объектно-ориентированных языков программирования.

Первый критерий означает пять свойств: стабильность (persistence), управление вторичной памятью, параллелизм, восстанавливаемость и средства обеспечения незапланированных запросов.

Второй означает восемь свойств: сложные объекты, идентифицируемость объектов, инкапсуляцию, типы или классы, наследование, перекрытие методов совместно с поздним связыванием, расширяемость и вычислительную полноту.

Необязательные возможности

Множественное наследование, проверка и вывод типов, распределенность, проектные транзакции (протяженные транзакции или вложенные транзакции), версии

Объектно-ориентированная база данных (ООБД)

* Объектно-ориентированная база данных (ООБД) — база данных, в которой данные моделируются в виде объектов, их атрибутов, методов и классов. Объектно-ориентированные базы данных обычно рекомендованы для тех случаев, когда требуется высокопроизводительная обработка данных, имеющих сложную структуру.

Преимуществами использования объектных БД перед реляционными являются:

* Отсутствует проблема несоответствия модели данных в приложении и БД (*impedance mismatch*). Все данные сохраняются в БД в том же виде, что и в модели приложения.
* Не требуется отдельно поддерживать модель данных на стороне СУБД.
* Все объекты на уровне источника данных строго типизированы. Рефакторинг объектно-ориентированной базы данных и работающего с ней кода теперь автоматизированный, а не однообразный и скучный процесс.

обязательные характеристики ООБД

В манифесте ООБД предлагаются обязательные характеристики, которым должна отвечать любая ООБД:

* Поддержка сложных объектов. В системе должна быть предусмотрена возможность создания составных объектов за счет применения конструкторов составных объектов. Необходимо, чтобы конструкторы объектов были ортогональны, то есть любой конструктор можно было применять к любому объекту.
* Поддержка индивидуальности объектов. Все объекты должны иметь уникальный идентификатор, который не зависит от значений их атрибутов.
* Поддержка инкапсуляции. Корректная инкапсуляция достигается за счет того, что программисты обладают правом доступа только к спецификации интерфейса методов, а данные и реализация методов скрыты внутри объектов.
* Поддержка типов и классов. Требуется, чтобы в ООБД поддерживалась хотя бы одна концепция различия между типами и классами.
* Поддержка наследования типов и классов от их предков. Подтип, или подкласс, должен наследовать атрибуты и методы от его супертипа, или суперкласса, соответственно.
* Перегрузка в сочетании с полным связыванием. Методы должны применяться к объектам разных типов. Реализация метода должна зависеть от типа объектов, к которым данный метод применяется. Для обеспечения этой функциональности связывание имен методов в системе не должно выполняться до времени выполнения программы.
* Вычислительная полнота. Язык манипулирования данными должен быть языком программирования общего назначения.
* Набор типов данных должен быть расширяемым. Пользователь должен иметь средства создания новых типов данных на основе набора предопределенных системных типов. Более того, между способами использования системных и пользовательских типов данных не должно быть

объектно-ориентированные системы управления базами данных (ООСУБД)

* Результатом совмещения возможностей (особенностей) баз данных и возможностей объектно-ориентированных языков программирования являются объектно-ориентированные системы управления базами данных (ООСУБД). ООСУБД позволяет работать с объектами баз данных так же, как с объектами в программировании в ООЯП. ООСУБД расширяет языки программирования, прозрачно вводя долговременные данные, управление параллелизмом, восстановление данных, ассоциированные запросы и другие возможности.
* Некоторые объектно-ориентированные базы данных разработаны для плотного взаимодействия с такими объектно-ориентированными языками программирования как *C#*, *C++*, *Visual Basic .NET*, *Java*, *Python*, *Objective-C* и *Smalltalk*; другие имеют свои собственные языки программирования. ООСУБД используют точно такую же модель, что и объектно-ориентированные языки программирования.

СУБД должна обеспечивать:

* Долговременное хранение
* Использование внешней памяти
* Параллелизм
* Восстановление
* Нерегламентированные запросы

* Объектная модель данных

В соответствии со стандартом ODMG 2.0 объектная модель данных характеризуется следующими свойствами.

* ▪ Базовыми примитивами являются объекты и литералы. Каждый объект имеет уникальный идентификатор, литерал не имеет идентификатора.

* ▪ Объекты и литералы различаются по типу. Все элементы одного типа имеют одинаковый диапазон изменения состояния (множество свойств) и одинаковое поведение (множество определенных операций). Объект, на который можно установить ссылку, называется экземпляром; он хранит определенный набор данных.

* ▪ Состояние объекта определяется набором значений, реализуемых множеством свойств. Этими свойствами могут быть атрибуты объекта или связи между объектом и одним или несколькими другими объектами.

* ▪ Поведение объекта определяется набором операций, которые могут быть выполнены над объектом или самим объектом. Операции могут иметь список входных и выходных параметров строго определенного типа. Каждая операция может также возвращать типизированный результат.

* ▪ База данных хранит объекты, позволяя совместно использовать их различным пользователям и приложениям. База данных основана на схеме данных, определяемой языком определения данных, и содержит экземпляры типов, определенных схемой.

* Объект, тип

Каждый тип имеет внешнюю спецификацию и одну или несколько реализаций. Спецификация определяет внешние характеристики типа: пользователю для работы с объектом предоставляется набор операций и набор атрибутов объекта, при помощи которых можно работать с реальными экземплярами. Реализация определяет внутреннее содержание объектов, например операции.

Тип также является объектом. Поддерживается иерархия супертипов и подтипов, реализуя стандартный механизм объектно-ориентированного программирования — наследование.

ООСУБД обслуживает множество баз данных, каждая из которых содержит определенное множество типов.

* Идентификатор объекта

Как это следует из модели данных, каждый объект в базе данных уникален. Существует несколько подходов для идентификации объекта. Самый простой — присвоить ему уникальный номер (OID — object identificator) в базе и никогда больше не повторять этот номер, даже если предыдущий объект с таким номером уже удален. Недостаток такого подхода состоит в невозможности перенести объекты в другую базу без потери связности между ними. Решение этой проблемы заключается в использовании составного идентификатора. Например, в Versant идентификатор OID имеет формат xxxxxxxx:yyyyyyyy, где xxxxxxxx — идентификатор базы данных, yyyyyyyy — идентификатор объекта в базе. Составленный таким образом OID позволяет переносить объекты из базы в базу без потери связи между объектами или без удаления объектов с перекрывающими номерами.

Идеальный вариант — использование OID, состоящего из трех частей: номер базы, номер класса, номер объекта. Однако и при этом остается вопрос о том, как обеспечить уникальность номеров баз и классов на глобальном уровне — при использовании ООСУБД на различных платформах, в разных городах и странах.

* Новые типы данных

Одним из принципиальных отличий объектных баз данных от реляционных является возможность создания и использования новых типов данных. Концептуально объект характеризуется поведением и состоянием. Определение типа заключается в определении поведения, т.е. операций, которые могут быть выполнены объектом или над состоянием объекта — набором атрибутов определенных типов (атрибут может иметь любой объявленный в базе тип). Важная особенность ООСУБД состоит в том, что создание нового типа не требует модификации ядра базы и основано на принципах объектно-ориентированного программирования: инкапсуляции, наследовании, перегрузке операций и позднем связывании.

Функционирование базы основано на схеме данных, которая может быть как первичной для создания классов или вторичной, выделяемой из созданных на языке программирования (C++) классов и загружаемой в базу. Язык ODL разработан ODMG как универсальный язык описания объектов. Для целей разработки предусмотрены элементы расширения классических объектных языков C++, Smalltalk, Java, позволяющих описать структуру объектов, их связи и типы связей.

* Оптимизация ядра СУБД

Ядро ООСУБД оптимизировано для операций с объектами. Естественными операциями для него являются кэширование объектов, ведение версий объектов, разделение прав доступа к конкретным объектам. Ядро объектно-реляционной СУБД остается реляционным, а «объектность» реализуется в виде специальной надстройки. Как следствие, ООСУБД свойственно более высокое быстродействие на операциях, требующих доступа и получения данных, упакованных в объекты, по сравнению с реляционными СУБД, для которых необходимость выборки связных данных ведет к выполнению дополнительных внутренних операций

* Язык СУБД и запросы

Общепризнанны две группы вариантов языков запросов.

* Язык OQL (Object Query Language) для объектных баз данных. Объектно-реляционные СУБД используют различные варианты объектных расширений SQL.

* Вторая группа языков запросов базируется на XML. Собирательное название языков этой группы — XML QL (или XQL). Они могут применяться в качестве языков запросов в объектных и объектно-реляционных базах данных.

* Транзакции

Короткие транзакции характеризуются малым временем выполнения; они могут существовать только в рамках сеанса работы с ООСУБД. Все изменяемые объекты блокируются, а после принятия транзакции разблокируются, изменения же записываются в базу данных.

Длинные транзакции предназначены для увеличения производительности при групповой работе. Можно создавать персональные и групповые базы. Пользователи работают со своей базой, а объекты из нее синхронизируются с групповой базой данных. Пользователь, начав длинную транзакцию, отмечает объекты, с которыми предстоит работать в групповой базе данных (операция «поставить на контроль» — check out). Эти объекты копируются в его персональную базу, а в групповой базе блокируются, причем блокировать их можно как на запись, так и на чтение. В групповой базе создается объект, содержащий все данные о длинных транзакциях. В случае повреждения групповой базы или физического отключения сервера групповой базы пользователь сможет продолжать работу с объектами в своей персональной базе, а после восстановления групповой базы — синхронизировать объекты. Перед завершением длинной транзакции пользователь должен поместить все измененные объекты обратно в основную базу (операция «зарегистрировать» — check in). После этого объекты копируются в основную базу, а блокировка снимается. В случае аварийного завершения длинной транзакции все изменения будут потеряны.

Вложенные транзакции по принципу функционирования аналогичны коротким. В процессе выполнения одной транзакции формируются другие. Если в текущем сеансе работает один процесс, то создается стек, а если несколько процессов — дерево транзакций.

* Блокировки

Назначение блокировок — гарантировать монопольность использования объекта конкретным пользователем с целью предотвращения одновременного изменения данных.

Короткие блокировки (short lock) предназначены для обеспечения последовательного доступа к данных при многопользовательском режиме работы.

Продолжительные блокировки (persistent lock) обеспечивают блокирование объектов на продолжительное время — часы, дни, недели. Применяются совместно с длинными транзакциями.

При этом объект может быть заблокирован несколькими способами:

- с исключением снятия другим процессом (hard lock);
- с возможностью снятия другим процессом (soft lock);
- по конкретным операциям.

* Перемещение объектов

Миграция объектов: постоянное их перемещение, например в другую базу данных. В качестве примера можно привести перемещение объектов из базы оперативных данных в базу данных архивного назначения.

Постановка на контроль (check out): копирование объектов в персональную базу данных при выполнении длинной транзакции.

Регистрация объектов (check in): копирование объектов в групповую базу данных из персональной при выполнении длинной транзакции.

Ведение версий

В «Манифесте объектно-ориентированных баз данных», поддержка множественных версий объектов отнесена к необязательным характеристикам ООСУБД. Однако большинство современных ООСУБД поддерживает версионность, что способствует повышению надежности информационной системы в целом.

*db4o

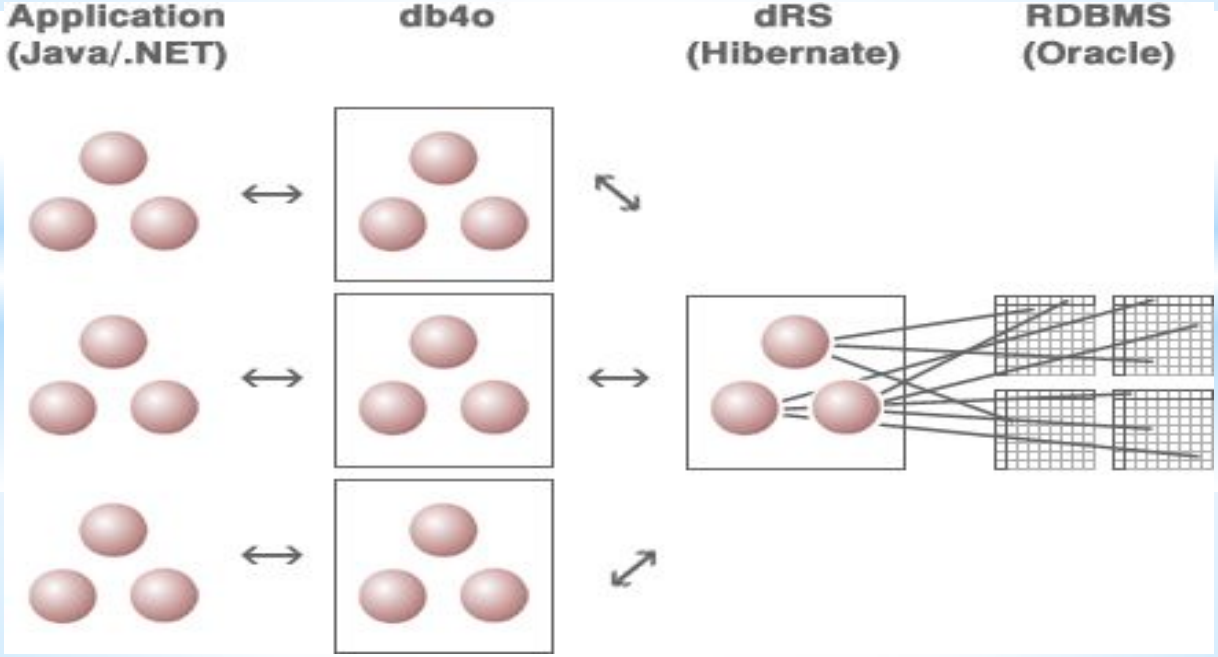
db4o — встраиваемая объектно-ориентированная СУБД с открытым исходным кодом, позволяющая *.NET* и *Java* разработчикам перманентно сохранять и получать объекты приложения с помощью одной строки кода.

Главный плюс — простота и прозрачность использования: необходимо просто добавить сборку *db4o* в своё приложение (*dll* или *jar* соответственно). Огромным плюсом является то, что для разработчика нет необходимости продумывать и реализовывать связку между объектной моделью данных и сущностями в базе — в качестве сущностей хранятся сами объекты.

db4o предоставляется с графический интерфейсом пользователя в виде плагина для *Visual Studio* или *Eclipse*.

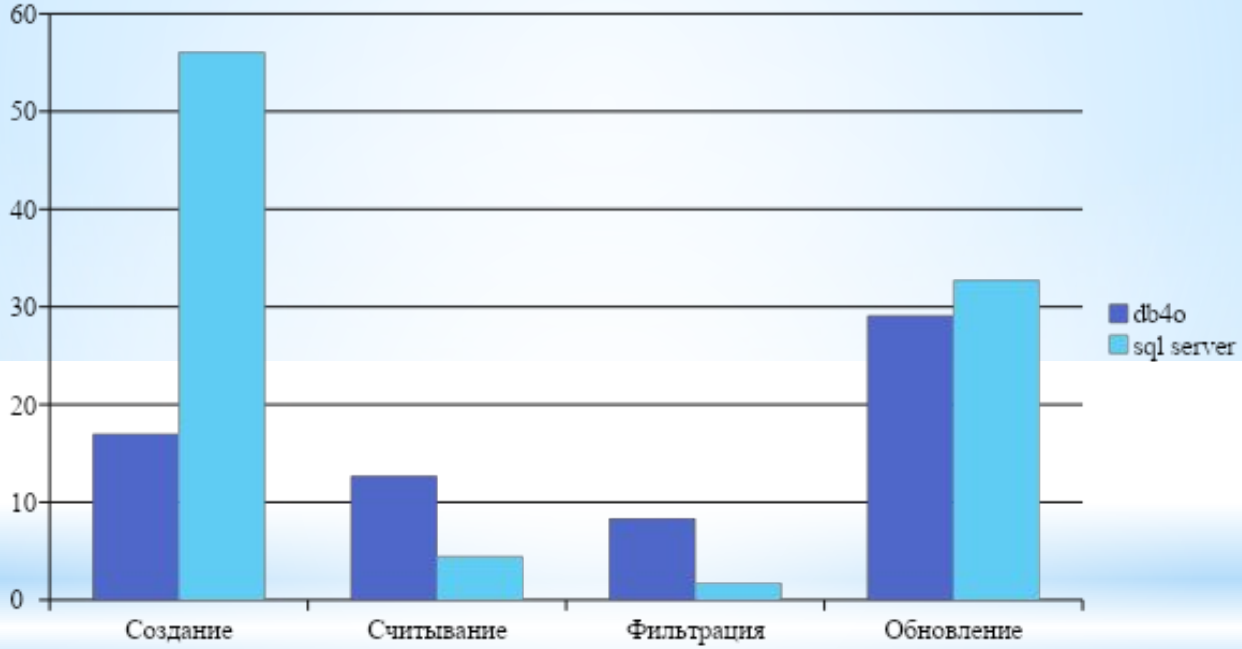
репликации *db4o*

db4o поддерживает *ACID*-транзакционность и репликации с помощью *db4o Replication System* (*dRS*) — специальной репликационной системы, разработанной компанией *Hibernate*. Репликация производится по равнозначным узлам, причём данные могут реплицироваться не только между двумя инстансами *db4o*, но и от *db4o* к реляционной базе, например *Oracle* или *MySQL*, однако здесь теряются такие плюсы объектной модели, как прозрачность и простота использования, так как в таком случае всё-таки придётся позаботиться о связке двух моделей; кроме того, произойдёт потеря производительности.



Производительность *db4o*

Производительность *db4o* показывает гистограмма. На ней показано время выполнения операций в сравнении с *MS SQL Server*. Сравнение производилось на объёме данных в 100 тыс. строк.



Как видно из гистограммы, *db4o* сильно выигрывает при создании (записи) элементов, но проигрывает при считывании и фильтрации.

*Objectivity/DB

- *Objectivity/DB* — объектно-ориентированная база данных, использующая иерархии, ассоциативные массивы, хеш-таблицы а также *STL*-контейнеры для манипулирования данными. Все объекты достаются по идентификатору объекта (*OID*). Существует две реализации *Objectivity/DB* — для *C++* и для *Java*.
- *Objectivity* поддерживает *ACID*-транзакционность и репликацию на равнозначные узлы с возможностью реплицирования на сервера, написанные на другом языке программирования и, соответственно, использующие другую сборку *Objectivity/DB*.

Использование Объектно-ориентированных баз данных

- *Объектно-ориентированные* базы данных хорошо подходят в том случае, когда объектная схема данных трудно переносится на *реляционную* или другую модель (например, в *документы*), при условии, что под используемую платформу существует объектная БД. Например, когда существуют множественные перекрёстные связи между классами.
- Они очень удобны благодаря своей прозрачности, то есть тому факту, что от разработчика не требуется разрабатывать прослойку между двумя различными моделями данных. ООБД можно воспользоваться для разработки приложений, использующих встроенную базу для хранения некоторых внутренних данных.
- Также, если объекты приложения имеют сложную внутреннюю структуру, содержащую множество других объектов, то в реляционной модели может понадобиться множество таблиц и множество связок между ними, что существенно понизит производительность при запросах с множественными *join*'ами. Кроме того, постоянно меняющаяся структура объекта также не подходит для реляционной модели, а для объектной не представляет проблемы.
- С другой стороны, объектно-ориентированные базы данных совершенно не подходят в том случае, если к базе необходимы частые незапланированные, т.н. «*ad hoc*» запросы, то есть выборка данных, не регламентируемая приложением.

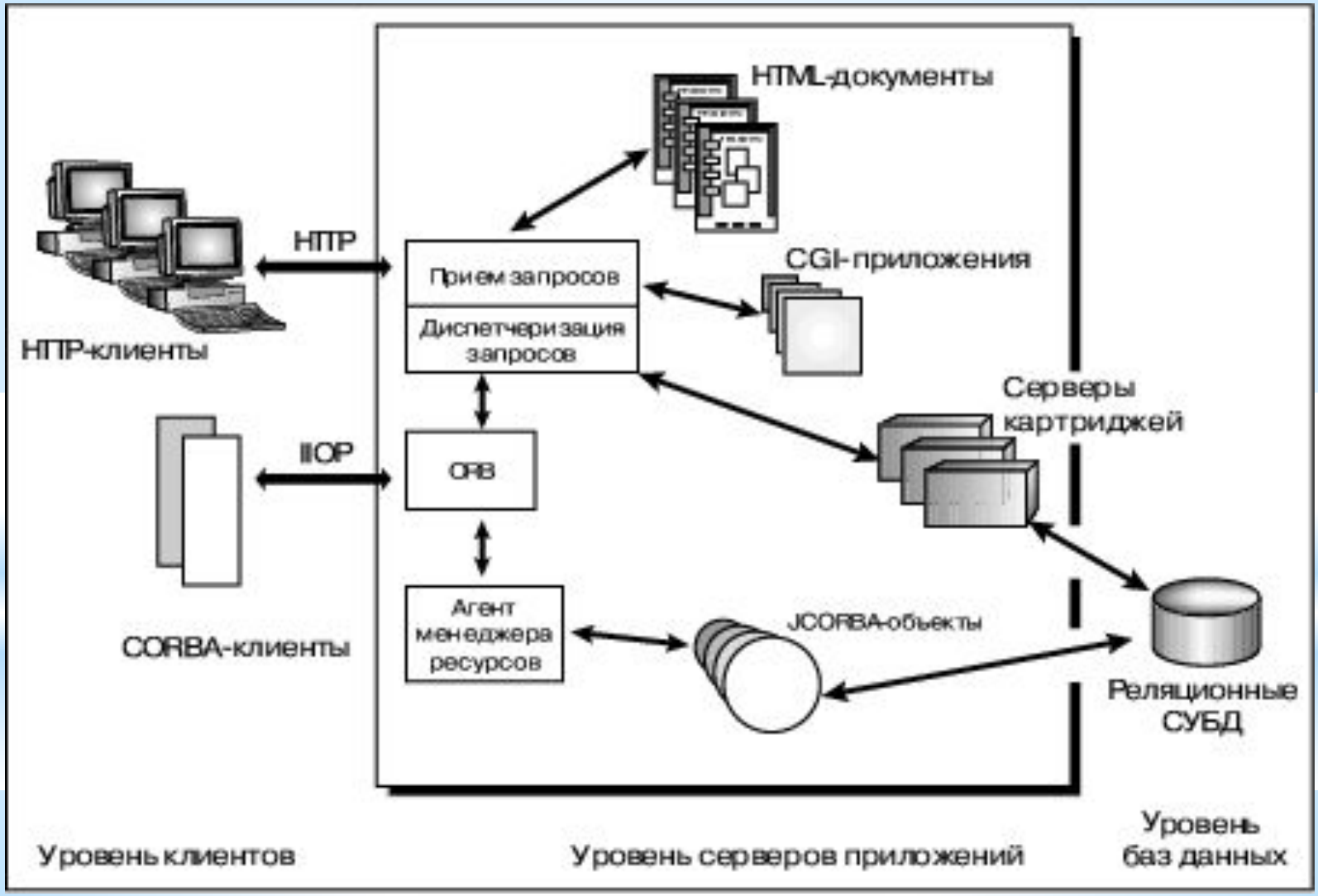
* Объектно-реляционные базы данных

В настоящее время применяется множество объектно-ориентированных языков программирования, а том числе C++ и Java. Такие языки дают возможность описывать объекты и манипулировать ими, однако имеют существенный недостаток – они не обеспечивают надежного и корректного хранения и считывания объектов. Тут то и нужны объектно-реляционные базы данных, подобные Oracle8. Система Oracle8 создана для хранения объектных данных и для работы с ними. Управление объектными данными аналогично управлению реляционными данными и осуществляется с помощью языка SQL, выступающего в роли средства взаимодействия с базами данных. В объектно-реляционной базе данных язык SQL (и PL/SQL) используется для манипулирования как реляционными, так и объектными данными. Кроме того, Oracle8 обеспечивает:

* Эффективное управление транзакциями;
* Надежное резервное копирование и восстановление информации;
* Высокопроизводительную обработку запросов;
* Блокирование данных;
* Параллельность работы пользователей;
* Расширяемость самой системы.

Объединение объектов с реляционной моделью даст отличные результаты – эффективность и надежность реляционной базы данных соединятся с гибкостью и средствами моделирования объектной структуры.

* Архитектура Oracle Application Server



СУБД Oracle9*i*

СУБД Oracle9*i* быстро превратилась в СУБД для всех типов данных – от простых до сложных. Мультимедийные типы данных, такие, как изображения, карты, видео- и аудио- клипы, редко обрабатывались неспециализированным программным обеспечением. Но в настоящее время многие веб-приложения требуют от своих серверов баз данных управления такими данными. Иные программные решения были также необходимы для хранения данных, которыми оперируют:

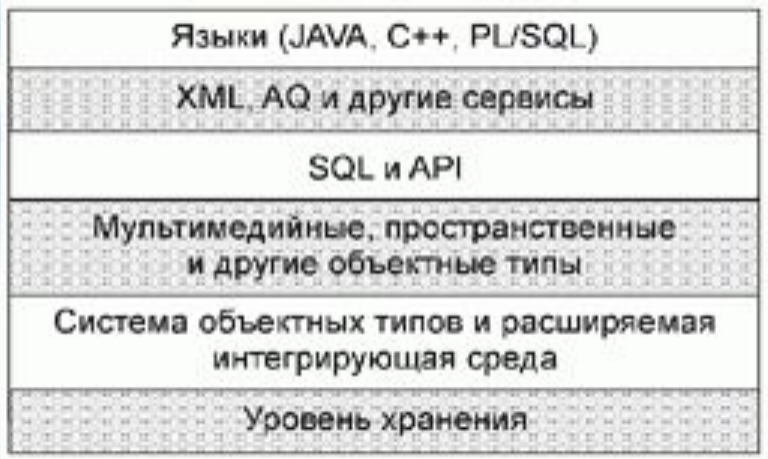
- финансовые инструменты;
- технические диаграммы;
- молекулярные структуры.

Для удовлетворения этих потребностей сервер баз данных Oracle9*i* предоставляет объектно-реляционную технологию, которая обеспечивает простые методы разработки, развертывания и управления приложениями, оперирующими со сложными данными.

* Объектно-реляционная архитектура СУБД Oracle9*i*

Сервер Oracle9*i* с объектно-реляционной технологией может быть "подогнан" разработчиками для создания их собственных специфических для области применения (application-domain-specific) типов данных. СУБД Oracle9*i* ™ была расширена для поддержки полных возможностей объектного моделирования, включая наследование (inheritance) и многоуровневые коллекции (multi-level collections), а также эволюции типов данных (type evolution). Например, можно создать новые типы данных, представляющие клиентов (customers), финансовые портфели (financial portfolios), фотографии и телефонные сети – и, тем самым, обеспечить, чтобы ваши приложения баз данных оперировали абстракциями, свойственными вашей предметной области (application domain).

Кроме того, весьма желательно интегрировать эти новые типы с сервером баз данных настолько тесно, насколько это возможно, чтобы они обрабатывались наравне со встроенными типами данных, такими, как NUMBER или VARCHAR.



* Объектно-ориентированная разработка приложений

СУБД Oracle9*i* предлагает большой набор интерфейсов прикладного программирования (API), реализующих связывания для различных языков. Для Java и PL/SQL предлагается "прямая" (native) поддержка внутри самой СУБД с тесной интеграцией между системой объектно-реляционных типов и хранимыми процедурами, написанными на Java или PL/SQL. Используя объектно-реляционную среду, можно хранить данные XML и эффективно манипулировать ими, индексировать их и эффективно обрабатывать запросы. Можно также поддерживать отображение между типами языка SQL и клиентских языков программирования (Java и C++), чтобы обеспечить "бесшовный" доступ к экземплярам типов данных SQL из приложений, написанных на Java или C++.

Индустриальные стандарты для разработки объектно-ориентированных приложений:

- UML (Unified Modeling Language) – унифицированный язык моделирования для объектно-ориентированного анализа и проектирования;
- стандарт объектно-реляционных баз данных SQL:1999;
- стандарты языков объектно-ориентированного программирования Java и C++.

Спецификации UML определяют стандартные конструкции для описания объектно-ориентированного программного обеспечения как объектной модели.

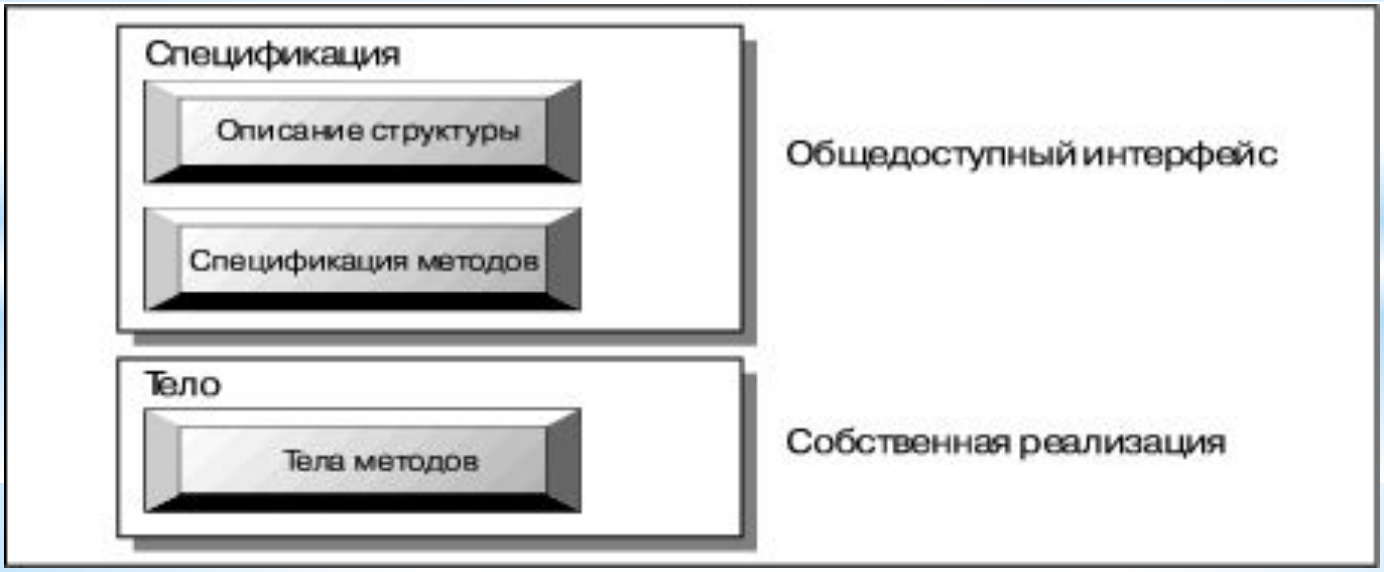
В 2003 г. консорциум OMG принял новую версию этого стандарта – UML 2.0.

* Объектные типы данных

Объектный тип данных — это тип, определяемый пользователем, и задающий как структуру (атрибуты), так и поведение (методы) объектов.

Разделение интерфейса и реализации относится к числу общих мест объектного подхода. Описание объектного типа состоит из двух частей. В интерфейсной декларируются атрибуты объектов и заголовки методов (процедур и функций). В теле типа приводится реализация методов.

Две части описания объектного типа данных



* Наследование

Наследование типов (Type inheritance) – это фундаментальная концепция в любой объектно-ориентированной системе. Наследование типов позволяет совместно использовать похожие свойства различных типов, а также расширять их характеристики.

Во многих объектно-ориентированных приложениях объекты организованы в типы, а типы – в иерархии типов. Эмпирически вполне достаточно организовать иерархии типов в виде набора деревьев. Тем самым, простого наследования достаточно для поддержки организации типов в большинстве приложений. Java – это объектно-ориентированный язык программирования, поддерживающий простое наследование. С помощью простого наследования тип может расширять один супертип (наследовать от одного супертипа). Такой тип, называемый подтипом (subtype), наследует все атрибуты и методы своего супертипа (supertype). Подтипу можно также добавлять новые атрибуты и методы или переопределять унаследованные методы. СУБД Oracle поддерживает модель простого наследования.

Типы-коллекции

Коллекции – это типы данных SQL, составляющие элементы которых представляют собой множественные элементы. Каждый элемент или значение для коллекции обладает тем же самым подстановочным типом данных. В Oracle предусмотрено два типа коллекций – массивы переменной длины (Varrays) и вложенные таблицы (Nested Tables).

Массив переменной длины содержит переменное число упорядоченных элементов. Типы данных VARRAY могут быть использованы для столбцов таблиц или атрибутов объектных типов.

С помощью Oracle SQL можно создавать указанные выше типы таблиц. Они могут использоваться как вложенные таблицы для реализации семантики неупорядоченной коллекции. Так же как и VARRAY, типы вложенных таблиц могут быть использованы для столбцов таблиц или атрибутов объектных типов.

Ссылочные типы

Если вы создаете объектную таблицу или объектное представление в СУБД Oracle9*i*, то можно получить ссылку (или указатель базы данных, *pointer*) на соответствующий объект-строку (row object). Ссылки важны для моделирования связей и навигации по экземплярам объектов, в частности, в приложениях на стороне клиента.

Большие объекты

СУБД Oracle9*i* предоставляет типы LOB (large object, большой объект) для решения проблем хранения изображений, видеоклипов, документов и других видов неструктурированных данных. Большие объекты хранятся таким образом, чтобы было оптимизировано использование пространства памяти и обеспечен эффективный доступ к ним. Конкретнее, большие объекты состоят из указателей (locators) и связанных с ними двоичных и/или символьных данных. Указатели этих объектов хранятся в строках таблиц вместе со значениями других столбцов.

Если применяются внутренние большие объекты (BLOB, CLOB и NCLOB), их данные размещаются в отдельной области хранения.

Для внешних же объектов (BFILE) их данные хранятся вне базы данных в файлах операционной системы.

* Связывания для языков программирования

Полная поддержка объектно-реляционной системы типов Oracle доступна в связываниях для ряда языков программирования, включая PL/SQL, Java и C/C++. К экземплярам типов можно получить доступ, и с ними можно манипулировать через интерфейсы прикладного программирования, такие, как JDBC (Java DataBase Connectivity) и OCCI (Oracle C++ Call Interface). Корпорация Oracle предоставляет также инструменты, подобные утилите JPublisher и транслятору объектных типов Object Type Translator (OTT), для отображения иерархий объектных типов в языки Java и C++. Кроме того, в средах этих языков также поддерживается подстановочность экземпляров и ссылок REF.

В СУБД Oracle 9i созданы новые и усовершенствованы существовавшие ранее механизмы обеспечения надежности и масштабируемости (Real Application Cluster, Logical Standby, FlashBack). Включена поддержка XML. В сервер Oracle интегрированы средства поддержки OLAP и добычи данных. Появился механизм Oracle Streams. Усовершенствованы средства управления, самонастройки и настройки.

* Поддержка XML, дуализм XML/SQL

Сервер Oracle поддерживает не только реляционную, объектную, многомерную модель данных, но и XML. Поддерживаются XML-схемы и XML-объекты: таблицы с типом XMLType и колонки типа XMLType.

Реляционные и XML-данные сосуществуют в одной универсальной модели. С XML-данными можно работать посредством языков SQL и Java, а с реляционными — через XML-интерфейсы, например, через XPath. Поскольку из SQL можно работать с XML-данными и их частями, то теперь легко построить обычный индекс по реквизиту, содержащемуся в XML-файлах и быстро находить нужные файлы. Можно построить реляционное представление (View), колонками которого будут реквизиты XML-файлов и далее работать с этим представлением обычными «реляционными» средствами. Можно написать запрос, одновременно работающий с реляционными данными, очередями сообщений, XML-данными, пространственными данными, контекстом. И наоборот, создав над реляционными или объектными таблицами базы данных представление XMLType View, можно работать с этими данными через XML-интерфейс.

* Поддержка OLAP

Реляционная модель удобна для представления данных в информационно-управляющих системах, однако для аналитических систем более подходит многомерная модель, где данные представлены в виде многомерных кубов, которые можно легко вращать, получать срезы, агрегировать информацию и т. д.

Для создания OLAP-приложений в Oracle ранее использовался программный продукт Express Server — СУБД с многомерной моделью. Данные из оперативных реляционных систем приходилось перегружать или подкачивать в Express Server, который не обеспечивал такого же уровня надежности, масштабирования, защиты, как реляционный сервер Oracle.

Сервер Oracle 9i поддерживает многомерную модель данных, что позволяет пользователю проектировать многомерные кубы и решать, как они будут храниться в Oracle 9i — в реляционных таблицах или в аналитических пространствах (LOB-поля). Обеспечивается возможность переноса данных из базы Express Server в Oracle 9i. Реализован весь набор функций, ранее присущий Express, причем разработчикам Oracle удалось добиться того, что скорость выполнения этих функций была не ниже, чем в Express Server. Алгоритмы добычи данных (data mining) встроены в сервер Oracle 9i.

* Oracle 10*g*

Oracle первой предложила СУБД, предназначенную для корпоративных сетей нового типа - систем распределенных вычислений (Grids).

Oracle 10*g* и Grid вычисления предоставляют предприятиям гибкость для удовлетворения меняющихся потребностей бизнеса, высокое качество услуг при небольших расходах, защиту инвестиций и их быструю окупаемость.

Помимо реализации на корпоративном уровне концепции Grid, новая платформа Oracle 10*g* предлагает 10 важнейших усовершенствований:

- рекордное повышение производительности
- самоуправляемость
- автоматическое управление хранением и доступом к данным (ASM)
- обновление ПО и приложений без остановки работы системы
- новые средства обеспечения высокой готовности
- упрощение установки и управления Oracle Real Application Clusters (RAC)
- быстрый перенос частей базы данных между разными платформами
- сокращение времени восстановления при сбоях с минут до секунд
- поддержка огромных баз данных - до 8 эксабайт (10**18)
- новые инструменты web-разработки HTML DB, развитие языка SQL.

* Oracle Database 10*g*

Oracle Database 10*g* предназначена для эффективного развертывания на базе различных типов оборудования, от небольших серверов до мощных симметричных многопроцессорных серверных систем, от отдельных кластеров до корпоративных распределенных вычислительных систем. СУБД предоставляет возможность автоматической настройки и управления, которая делает ее использование простым и экономически выгодным. Ее возможности осуществлять управление всеми данными предприятия - от обычных операций с бизнес-информацией до динамического многомерного анализа данных (OLAP), операций с документами формата XML, управления распределенной/локальной информацией - делает ее идеальным выбором для выполнения приложений, обеспечивающих обработку онлайновых транзакций, интеллектуальный анализ информации, хранение данных и управление информационным наполнением.

* Oracle Application Server 10*g*

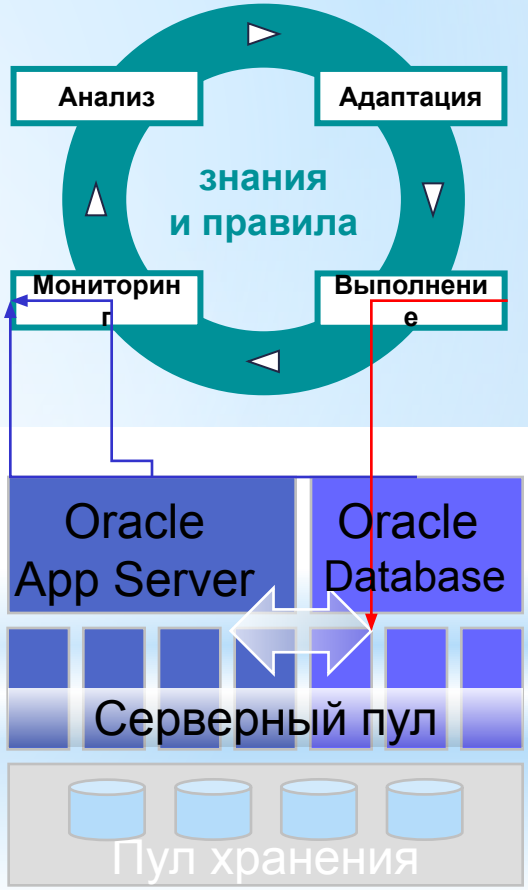
Oracle Application Server 10*g* - это основанная на стандартах интегрированная программная платформа, позволяющая организациям любого масштаба оперативнее реагировать на меняющиеся требования рынка. Oracle Application Server 10*g* обеспечивает полную поддержку технологии J2EE и распределенных вычислений, включает встроенное ПО для корпоративных порталов, высокоскоростного кэширования, интеллектуального анализа бизнес-данных, быстрого развертывания приложений, интеграции бизнес-приложений, поддержки беспроводных технологий, Web-сервисов - и все это в одном продукте. Поскольку платформа Oracle Application Server 10*g* оптимизирована для Grid Computing, она позволяет повысить степень готовности IT-систем и снизить расходы на приобретение аппаратных средств и администрирование.

Oracle Enterprise Manager 10g

Oracle Enterprise Manager 10*g* - это первое в отрасли программное обеспечение, разработанное для администрирования корпоративных сетей распределенных вычислений на базе решений Oracle. Оно призвано помочь снизить сложности, сопряженные с администрированием бизнес-приложений, благодаря управляющему ПО, которое позволяет получить полную информацию обо всей вычислительной инфраструктуре компании. Оно дает системным администраторам возможность реализовать политики, управлять уровнями обслуживания и перераспределять вычислительные ресурсы и приложения при изменении требований бизнеса. Oracle Enterprise Manager 10*g* построено на базе открытой основанной на стандартах архитектуры. Оно поддерживает ключевые стандарты управления, разработанные комитетом Distributed Management Task Force (DMTF), включая Common Information Model (CIM) и Web-based Enterprise Management (WBEM).

Адаптивная платформа для Oracle

- ASCC для Oracle – адаптивная инфраструктурная платформа для приложений и баз данных «по требованию»
- Предложение включает в себя
  - предварительно сконфигурированные серверы, системы хранения и программные компоненты, обеспечивающие автоматизацию, виртуализацию и развёртывание
  - службы для интеграции стандартных и индивидуальных приложений на базе Grid-технологий Oracle Database Server и Oracle Application Server

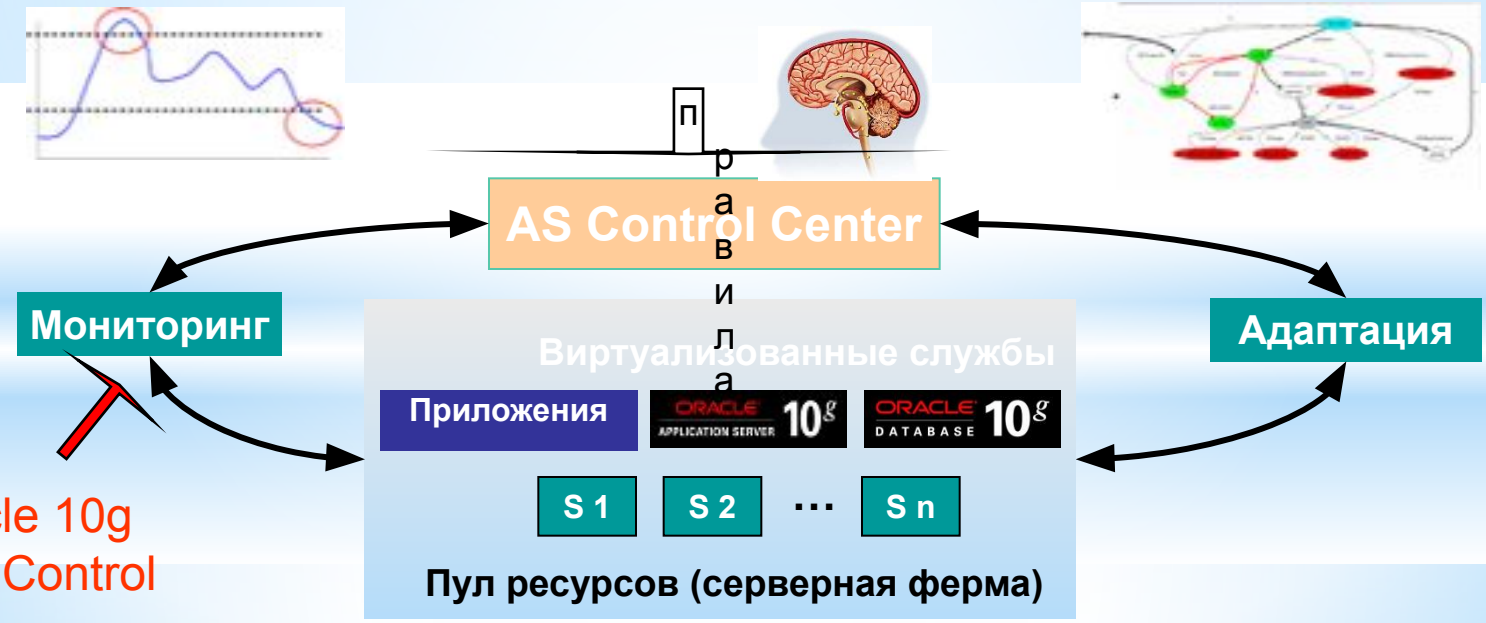


* Adaptive Services Control Center (ASCC)

Непрерывный мониторинг физических и виртуальных ресурсов

Ресурсы распределяются адаптивно (в зависимости от загрузки систем)

Для адаптации используются произвольно назначаемые правила и методы



* Сценарий: управление загрузкой на базе правил

Grid Control (мониторинг)

загрузка CPU

время отклика

сигнал:медленный отклик

сигнал:высокая загрузка CPU на AS2

ASCC (анализ, адаптация)

правил а

развёртывание AS3

Серверы

AS1

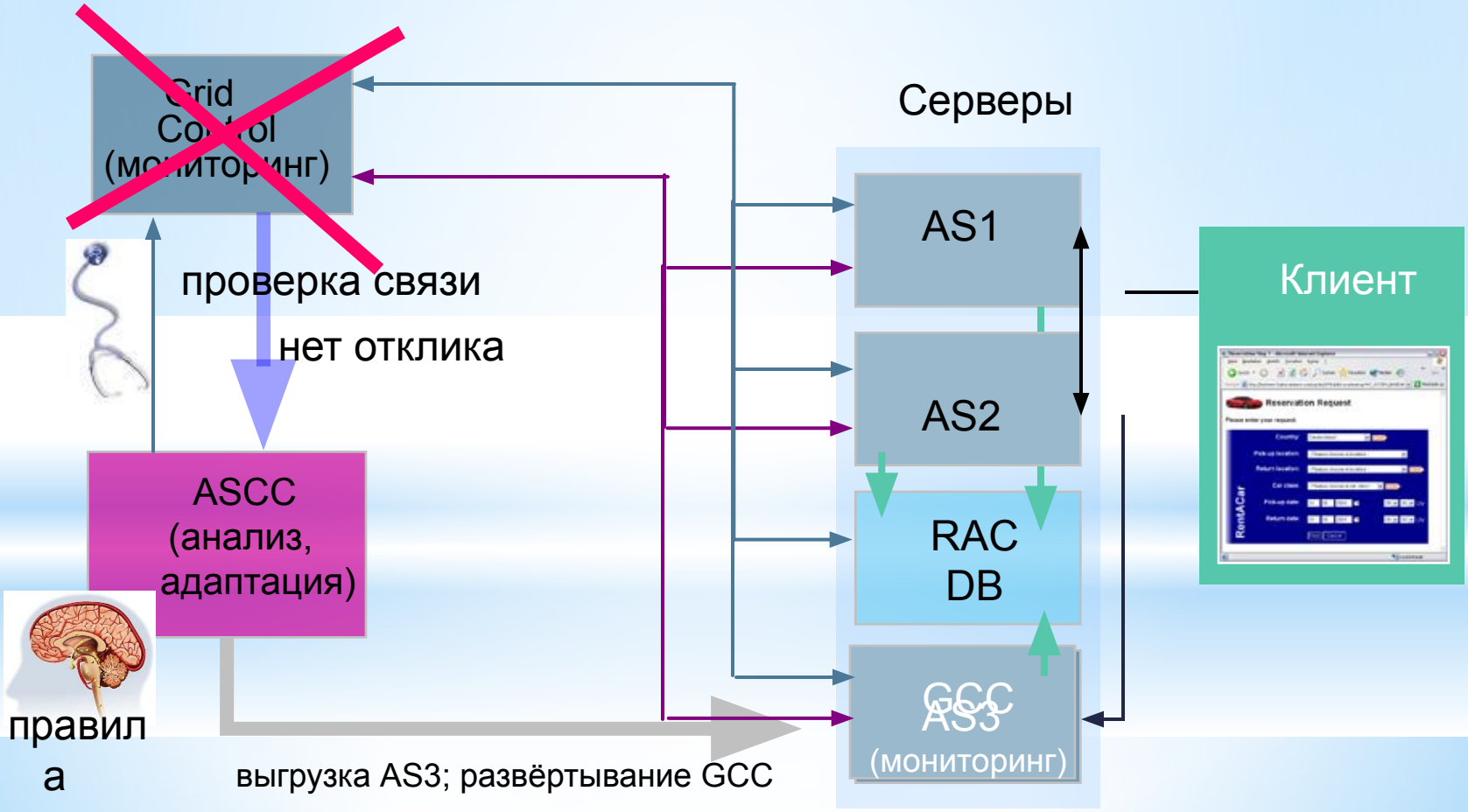
AS2

БД RAC

резерв AS 3

Клиент

*Сценарий: преодоление сбоя



* Недостатки SQL-хранилищ

* Скорость работы. При высокой загруженности ресурса и достаточно большом количестве записей в базе данных — начинаются проблемы:

- индексирование и операции слияния таблиц (*join*) работают очень медленно;
- требуется большое количество дискового пространства, оперативной памяти и ресурсов ЦПУ.

* Масштабируемость. Реляционная модель с этой задачей справляется не лучшим образом.

* Атрибуты (столбцы) всегда типизированы, то есть мы не имеем возможности хранить данные с различными типами атрибутов в одной таблице. Кроме того, набор атрибутов кортежа всегда фиксирован, таким образом, мы не можем хранить сущности разных форматов (с различным набором атрибутов) в одной строке.

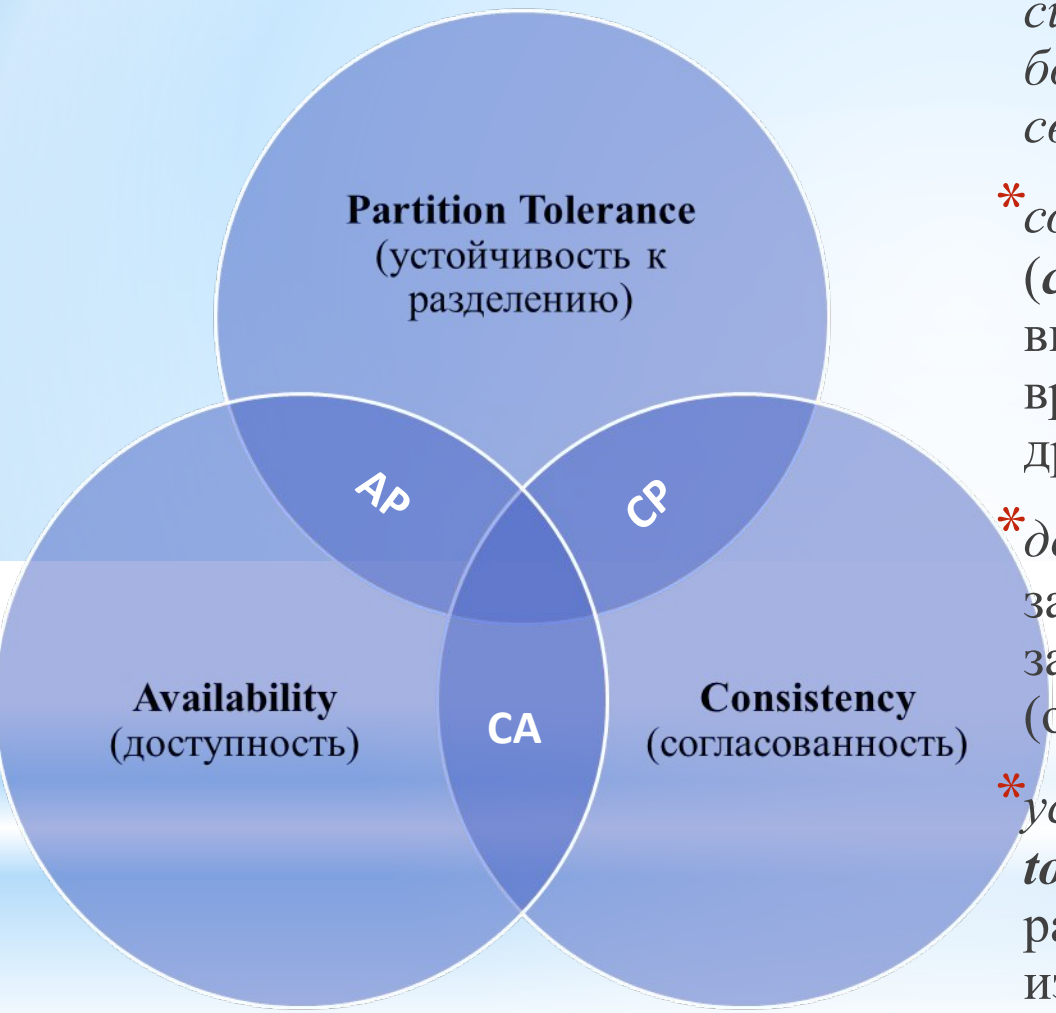
* Сложность SQL. Для управления данными требуется знание языка, а при достаточно сложной структуре базы — умение писать запросы соответствующей сложности.

* Всё это ведёт к тому, что разработчикам больших хранилищ данных необходимо искать пути реализации своих проектов, отличные от стандартных реляционных моделей.

* Нереляционный подход к организации БД - NoSQL

NoSQL хранилища данных имеют максимально упрощённую структуру «ключ — значение» и набор *CRUD* операций (*create*, *read*, *update*, *delete*), что обеспечивает отличную масштарибуемость и скорость. Под «значением» здесь понимается всё что угодно — от строки до файла или даже ещё одной таблицы. *NoSQL* — движение, объединяющее в себе совершенно разные проекты нереляционных СУБД. Отличительные черты *NoSQL* — нереляционные модели данных, простые *API* или протоколы доступа, способность к горизонтальному масштабированию по требованию для некоторого набора операций на многих серверах, распределенное хранение данных, эффективное использование распределенных индексов и памяти для запросов, достаточно свободное обращение с такими незыблемыми для традиционных СУБД вещами, как транзакционная целостность.

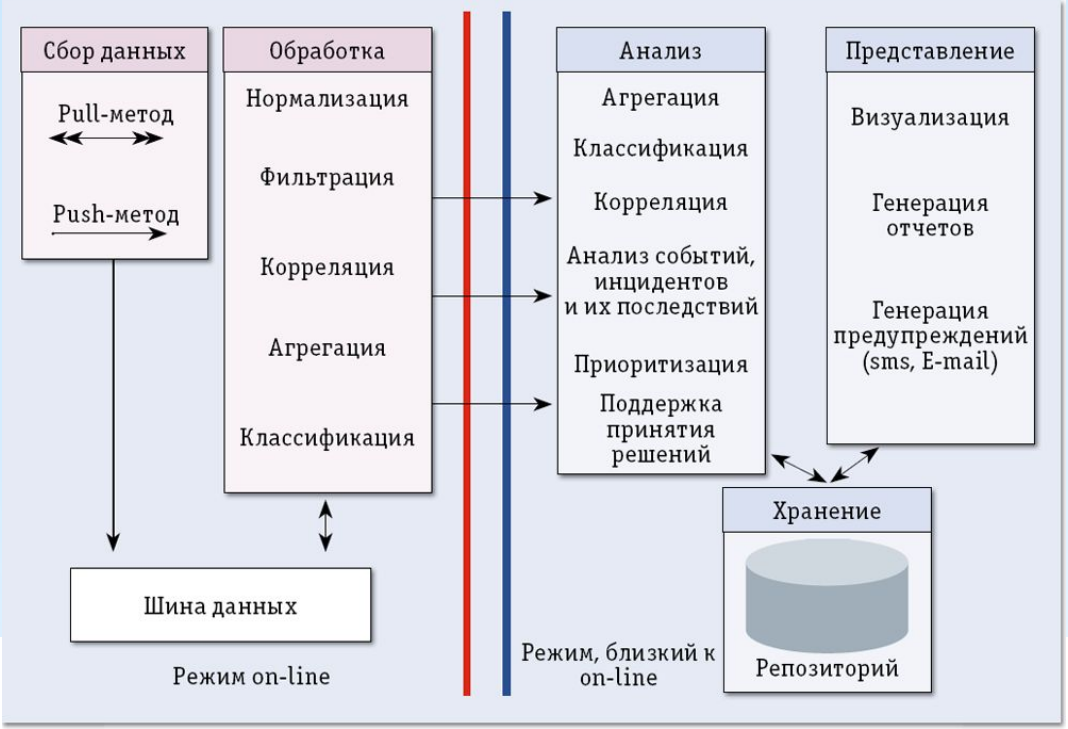
Теорема CAP (Эрик Брюер)



В *распределенной вычислительной системе невозможно обеспечить более двух из трёх следующих свойств*:

- *согласованность данных* (***consistency***) — во всех вычислительных узлах в один момент времени данные не противоречат друг другу;
- *доступность* (***availability***) — любой запрос к распределённой системе завершается корректным откликом (ошибкой);
- *устойчивость к разделению* (***partition tolerance***) — расщепление распределённой системы на изолированные секции не приводит к некорректности отклика от каждой из секций

* Выполнение запроса к распределенной системе



Пусть распределенная система состоит из *N* серверов, каждый из которых обрабатывает запросы некоторого числа клиентских приложений. При обработке запроса сервер должен гарантировать актуальность информации, содержащейся в отсылаемом ответе на запрос, для чего предварительно нужно выполнить синхронизацию содержимого его собственной базы с другими серверами. Таким образом, серверу необходимо ждать полной синхронизации либо генерировать ответ на основе не синхронизированных данных. Возможен и третий вариант, когда по каким-либо причинам синхронизация производится только с частью серверов системы.

В первом случае оказывается не выполненным требование по доступности, во втором — по согласованности, в третьем — по устойчивости к разделению.

* Классификация распределенных систем по видам выполняемых требований *CAP*

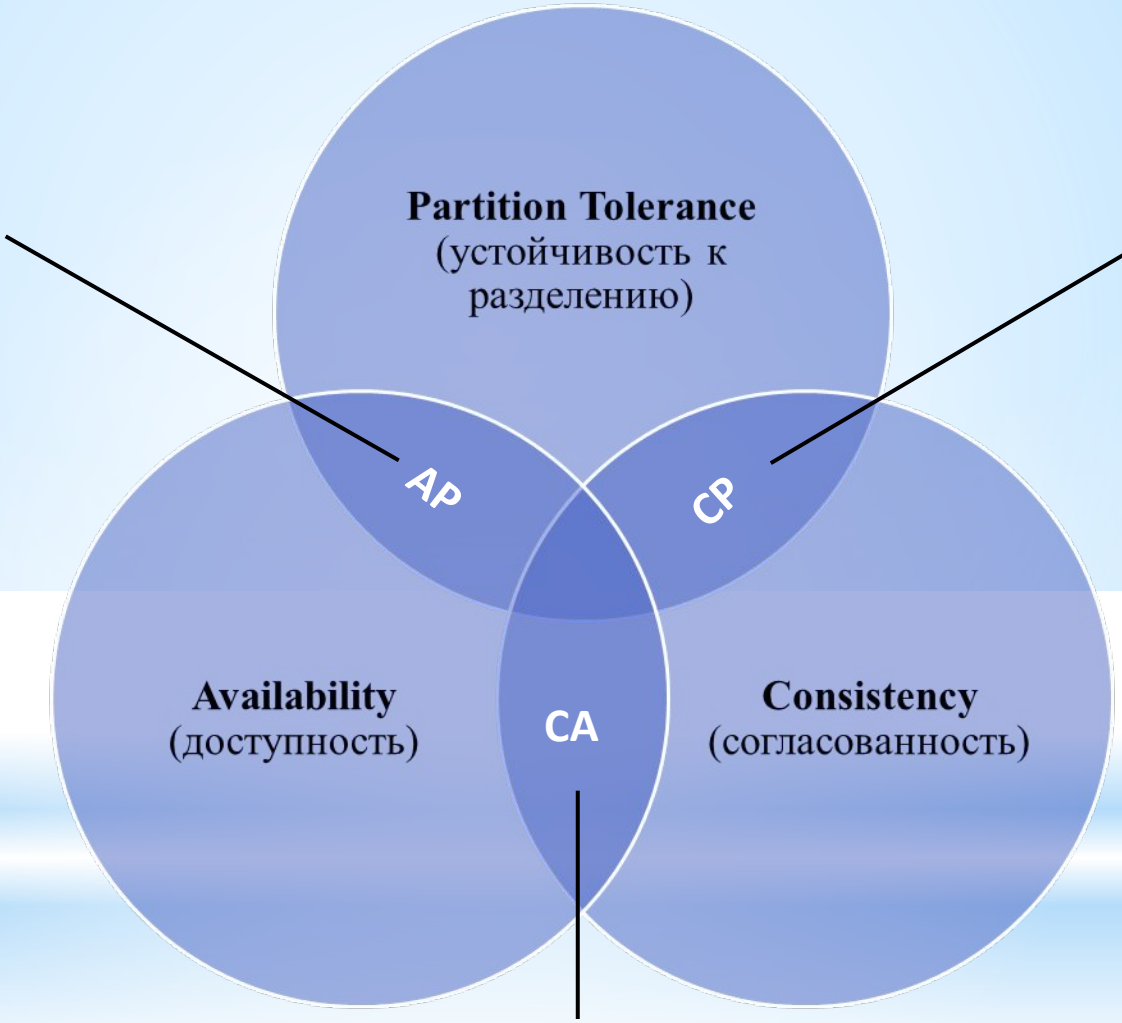
* ***CA*** — система, во всех узлах которой данные согласованы и обеспечена доступность, жертвует устойчивостью к распаду на секции. Такие системы возможны на основе технологического программного обеспечения, поддерживающего транзакционность в смысле *ACID*. Примерами таких систем могут быть решения на основе кластерных систем управления базами данных или распределённая служба каталогов *LDAP*;

* ***CP*** — распределённая система, в каждый момент обеспечивающая целостный результат и способная функционировать в условиях распада, в ущерб доступности может не выдавать отклик. Устойчивость к распаду на секции требует обеспечения дублирования изменений во всех узлах системы, в этой связи отмечается практическая целесообразность использования в таких системах распределённых пессимистических блокировок для сохранения целостности;

* ***AP*** — распределённая система, отказывающаяся от целостности результата. Задачей при построении *AP*-систем становится обеспечение некоторого практически целесообразного уровня целостности данных, в этом смысле про *AP*-системы говорят как о *«согласованности в конечном счёте»*.

* Типы СУБД с точки зрения *CAP*

BerkeleyDB
Voldemort
Redis
Cassandra
CouchDB



HBase

РСУБД (MSSQL, MySQL, Oracle),

Neo4j , db4o

* BASE (Basically Available, Soft State, Eventually consistent)

* В любом из трех случаев не обязательно будет выполнено свойство ***ACID*** (*Atomicity*, *Consistency*, *Isolation*, *Durability* — «атомарность, согласованность, изолированность, долговечность»), обычно строго соблюдаемое в реляционных СУБД, а ему противопоставляется свойство ***BASE*** (*Basically Available*, *Soft State*, *Eventually consistent*) — базовая доступность, неустойчивое состояние, согласованность в конечном счёте. При сбое в некоторых узлах системы, отказ получает только часть приложений, взаимодействующих с вышедшими из строя узлами. В ходе взаимодействия используются протоколы без состояния, что снижает нагрузку на отдельные узлы и позволяет ее перераспределять. Наконец, допустима временная несогласованность данных в разных узлах системы при условии, что информация будет синхронизирована через некоторый обозримый промежуток времени.

* *BASE* используется для наиболее общего описания требований к распределенным *NoSQL*-системам, подпадающих под утверждение теоремы *CAP* и не удовлетворяющих требованиям *ACID*.

Масштабируемость

- Одной из основных проблем реляционных баз данных, решение которых может быть достигнуто с помощью технологий *NoSQL*, является масштабируемость. Она представляет собой способность системы справляться с увеличением нагрузки посредством наращивания её ресурсов.
- Основные подходы к масштабированию:
- **Вертикальное масштабирование**: увеличение производительности системы за счёт использования более мощного оборудования;
- **Горизонтальное масштабирование**: необходимый уровень производительности достигается посредством разбиения системы на структурные компоненты и распределения их по отдельным физическим машинам.
- Горизонтальное масштабирование — более сложное решение, но оно имеет целый ряд преимуществ: гибкость, относительно низкая стоимость и высокий потенциал для дальнейшего увеличения кластера. В контексте баз данных этот подход заключается в распределении данных между несколькими базами, находящимися на различных узлах кластера. К основным способам горизонтального масштабирования баз данных относятся репликация и шардинг

Репликация

* В распределенных базах данных репликация состоит в хранении одних и тех же данных на нескольких узлах. Это позволяет увеличить пропускную способность чтения данных, поскольку операции чтения могут быть распределены между множеством машин. Кроме того, это повышает отказоустойчивость системы.
* С другой стороны, репликация имеет недостаток, связанный с операциями записи. Каждая из таких операций должна быть выполнена для каждого узла, предназначенного для дублирования данных.

При этом возможны следующие варианты:

* Данные становятся доступными только после записи на каждый из узлов;
* После поступления данные становятся доступными вне зависимости от того, были ли они записаны на все узлы.
* В зависимости от выбранного варианта приходится пожертвовать либо доступностью, либо согласованностью данных.

Шардинг (Sharding)

* Другим подходом к горизонтальному масштабированию является шардинг (Sharding). Он представляет собой разбиение данных в базе на отдельные порции и распределение их по узлам. Для реляционных баз данных при шардинге логически независимые записи таблицы хранятся на различных узлах. Таким образом, на разных серверах будет находиться таблица с одинаковой структурой, но разными данными. Для осуществления распределения данных по узлам может быть использована хеш-функция, применяемая к первичному ключу записи для определения сервера, на котором она должна быть размещена. Чем больше узлов используется в шардированной базе данных, тем выше вероятность того, что один из них выйдет из строя. Поэтому шардинг обычно используется в сочетании с репликацией, что позволяет повысить отказоустойчивость системы.

* Основным преимуществом шардинга является возможность добавления и удаления узлов из кластера по мере изменения нагрузки без необходимости каких-либо изменений в приложении. Однако есть и существенный недостаток: некоторые операции, типичные для баз данных, становятся слишком сложными и медленными. Одна из наиболее важных операций для реляционных баз данных - операция соединения, служащая для материализации логической связи между данными базы. Результатом операции является отношение (таблица), получаемая как декартово произведение исходных отношений.

Шардинг (Sharding)

* В распределенных системах оба исходных набора данных хранятся на множестве узлов, и для осуществления соединения потребуется выполнить большое количество запросов к каждому из этих узлов, что вызовет значительный рост сетевого трафика. По этой причине в базах данных с возможностью шардирования обычно не поддерживается операция соединения.
* Сложности с распределенным выполнением операции соединения делают проблематичным горизонтальное масштабирование реляционных баз данных. Причиной является то, что многие существующие СУБД разрабатывались без расчета на использование горизонтального масштабирования - эта возможность была добавлена в готовые системы позднее.
* В то же время большинство *NoSQL*-баз данных изначально включают шардинг как одну из ключевых возможностей. Некоторые из них даже поддерживают автоматическое партиционирование и балансировку нагрузки на узлы кластера.

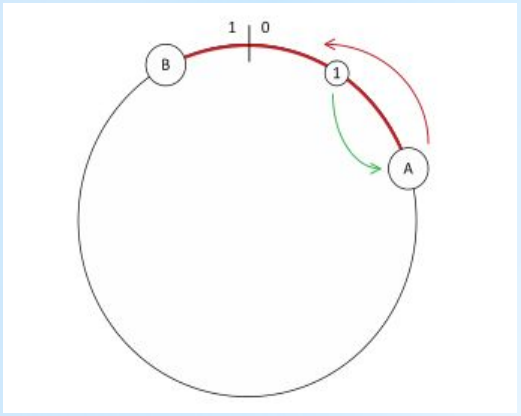
*Консистентное хеширование

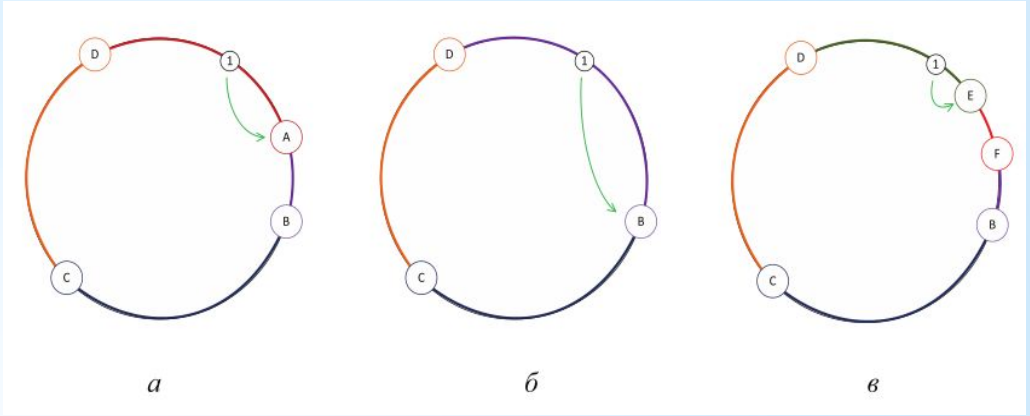
Для многих *NoSQL*-СУБД шардинг представляет собой одну из ключевых возможностей. При этом важной задачей является выбор механизма для определения узла, на котором должна быть размещена та или иная запись. Очевидным решением является применение простой хеш-функции к ключу записи с последующим делением результата на количество узлов в кластере:

$$index = hash(key) \bmod_n,$$

где *index* - индекс целевого узла, *key* - ключ записи, *n* - число узлов в кластере. Проблема такого подхода в том, что при добавлении/удалении хотя бы одного узла потребуется перераспределение всех данных в кластере, ведь значение *index* для любой записи может стать иным. Решением проблемы является механизм консистентного хеширования, используемый в некоторых *NoSQL*-базах данных, таких как *Amazon Dynamo*, *Project Voldemort* и *MemcacheDB*.

Добавление и удаление узлов





* Идея состоит в присвоении узлам кластера идентификаторов из области значений хеш-функции. Если представить область значений функции в виде окружности, каждый узел отвечает за дугу окружности (соответствующую некоторому интервалу значений хеш-функции) между идентификаторами самого узла и его ближайшего соседа против часовой стрелки. Таким образом, чтобы получить доступ к объекту с ключом *Key* необходимо обратиться к узлу, ближайшему по часовой стрелке от *Key*.
* При таком подходе изменения зон ответственности при добавлении/удалении компьютеров затрагивают наименьшее число узлов:
* Узел берет на себя ответственность за интервал значений, принадлежащий его ближайшему соседу против часовой стрелки, если тот покидает кластер (б);
* Появившийся в кластере новый узел принимает на себя ответственность за ближайший против часовой стрелки интервал (его ближайший сосед по часовой стрелке, соответственно, теряет контроль над этим интервалом) (в).

MapReduce

* Одним из наиболее популярных подходов к распределенной обработке данных является модель *MapReduce*, разработанная *Google*. Использование этой парадигмы позволяет программисту избежать необходимости написания специального кода для распараллеливания и синхронизации.

MapReduce предполагает разбиение задачи на два шага:

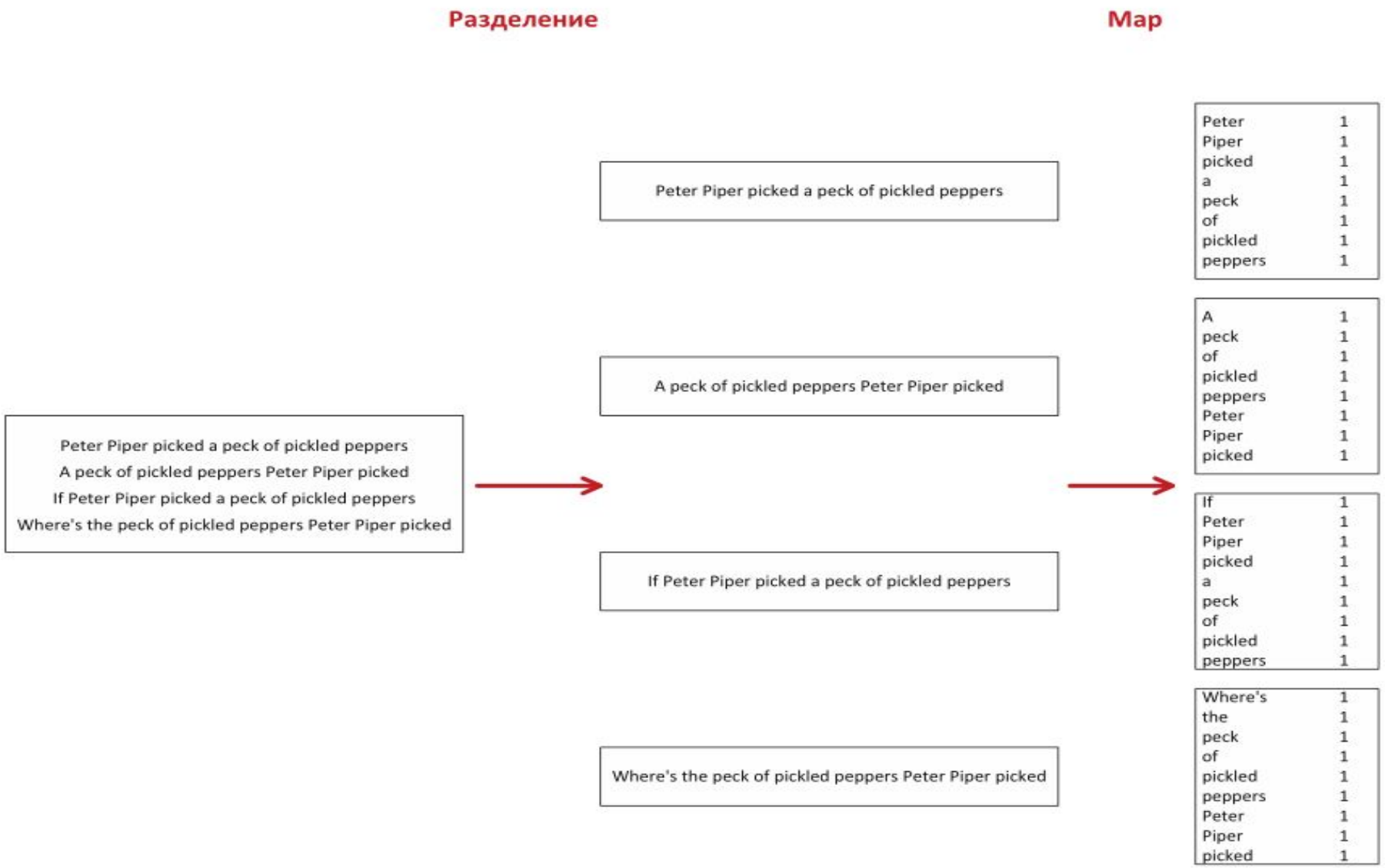
* *Map-шаг:* главный узел разбивает входные задачи на части и передает остальным узлам для предварительной обработки;

* *Reduce-шаг:* свертка результатов предварительной обработки. Главный узел получает данные от остальных узлов и формирует окончательный результат выполнения задачи.

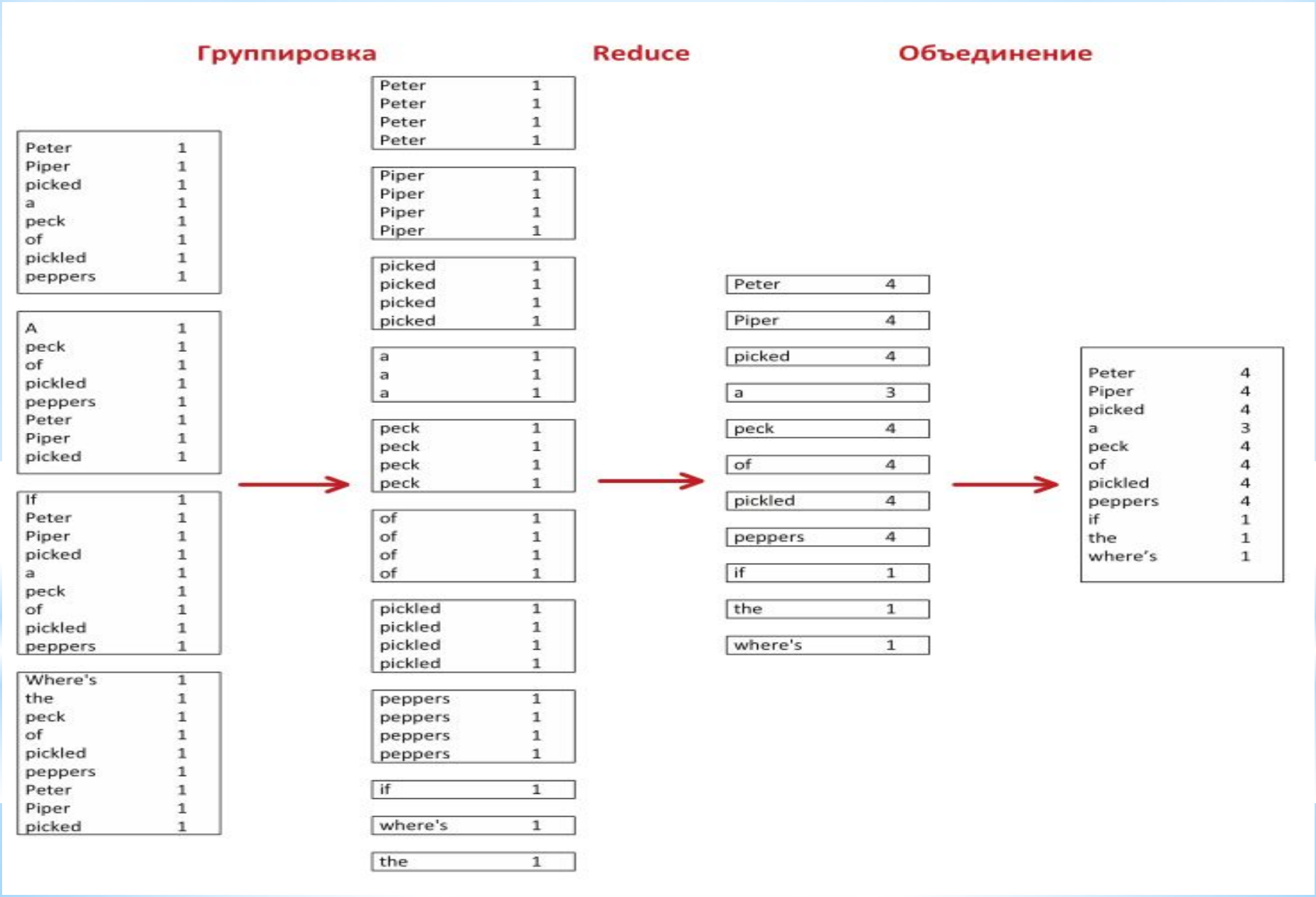
* Для использования *MapReduce* необходимо написать функции *Map* и *Reduce*. Эти функции работают с данными, структурированными в виде пар «ключ-значение». *Map* принимает на вход пару одного типа и возвращает набор пар другого типа (промежуточный результат): *Map*(*Key*,*Value*) □ *list*(*iKey*,*iValue*),

* где *(Key,Value)* - одна запись из набора входных данных, *(iKey,iValue)* - промежуточная пара ключ/значение. Каждая пара, сгенерированная *Map*-функцией, отправляется на некоторый узел для выполнения *reduce*-шага. Целевой узел определяется посредством применения хеш-функции к ключу пары *(iKey)*. На каждом узле полученные пары группируются по ключу. Затем к каждой группе применяется функция *Reduce,* при выполнении которой формируется окончательный результат *f Value*: *Reduce*(*iKey*, *list* (*iValue*)) □ *list* (*fValue*)

* Разделение и *Map* для подсчёта слов в тексте

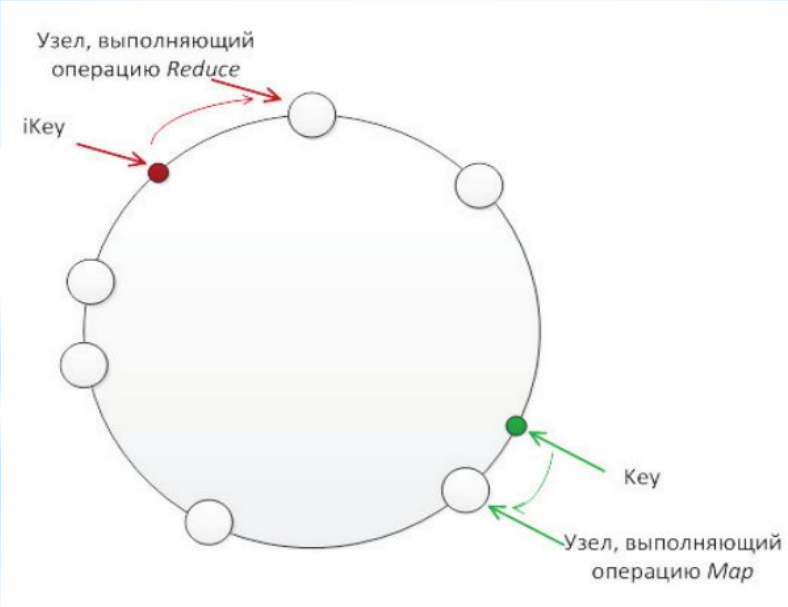


* Группировка и *Reduce* для подсчёта слов в тексте



Внедрение модели *MapReduce*

Модель *MapReduce*, предоставляющая широкие возможности для распределенной обработки данных, внедрена в ряд *NoSQL*-баз данных, таких как *MongoDB* и *CouchDB*. В СУБД для определения узлов, выполняющих операции *Map* и *Reduce* для пар *(Key, Value)* исходных данных и *(iKey,iValue)* промежуточных результатов соответственно, может быть использована консистентная хеш-функция. Возможность узлов самостоятельно определять, куда следует передать данные для последующей обработки, снимает потребность в узле-координаторе.



MapReduse с использованием консистентной функции

*Классификация *NoSQL*

В зависимости от направленности и способов обработки данных можно различать такие нереляционные хранилища как:

* Хранилища типа «**ключ-значение**»
* **Колоночные** хранилища (***Wide Column***)
* **Документно-ориентированные** хранилища
* **Объектно-ориентированные** хранилища
* Хранилища **графов**
* И множество более специализированных хранилищ.

* Хранилища типа «ключ-значение»

* Хранилища «ключ-значение» («*Key-Value*») — базы данных, построенные на основе простой модели ассоциативного массива, или хеш-таблицы, при которой пользователь записывает данные в базу и извлекает их из неё по соответствующему ключу. Если представлять себе такую таблицу в рамках реляционной СУБД, то она будет состоять всего из двух столбцов: **ключа** и **значения**, причём содержимое значения может быть абсолютно неважно — СУБД перекладывают ответственность за то, что находится в «значении» на приложение, которое пользуется хранилищем.

* Такие хранилища имеют базовый набор операций — ***CRUD*** — *Create* (запись), *Read* (чтение), *Update* (изменение), *Delete* (удаление). Манипулирование данными происходит с помощью ключа, всегда являющегося первичным, что обеспечивает отличную производительность и масштабируемость. Вообще, современные хранилища «ключ-значение» в основном отдают предпочтение масштабируемости в ущерб согласованности. Также они обычно не предоставляют широких возможностей аналитической обработки данных.

BerkeleyDB

- *Oracle BerkeleyDB* (*BDB*) — высокопроизводительная встраиваемая база данных, реализованная в виде библиотеки. *BDB* может обслуживать тысячи процессов или потоков, одновременно манипулирующих базами данных размером в 256 терабайт, на разнообразном оборудовании под различными операционными системами, включая большинство *UNIX*-подобных систем и *Windows*, а также на операционных системах реального времени.
- В *BDB* отсутствует сетевой доступ, манипулирование данными происходит с помощью *API*, существующего для множества языков программирования на большинстве платформ (в т.ч. *C#*, *C++*, *Java*). Поддерживает *ACID*-транзакционность, блокировки, репликацию и даже *SQL* в качестве одного из интерфейсов. *BerkeleyDB* не поддерживает разделение, каждый сервер репликаций хранит копию одних и тех же данных.
- Данные в *BerkeleyDB* хранятся в хеш-таблицах. Поиск производится с помощью бинарных деревьев. И ключи, и значения могут быть произвольными байтовыми строками, фиксированной или произвольной длины. Они могут хранить до 4ГБ информации (одна запись может хранить изображения, аудио, видео и другие данные большого размера.

*Dynamo u Voldemort

* В основу СУБД от *Amazon* положен принцип «сбой оборудования - стандартный режим работы системы», поэтому особое внимание при разработке было уделено отказоустойчивости. Является *AP*-системой. *Dynamo* использует алгоритм партиционирования с использованием консистентного хеширования. При записи данных осуществляется их репликация на несколько ближайших узлов. Поддерживается *ACID*, но полной транзакционности нет.
* В отличие от большинства других *NoSQL* СУБД, не является проектом с открытым исходным кодом. *Dynamo* распространяется по модели *Database as a Service*, как часть *Amazon Web Services*.
* *Project* ***Voldemort*** *— Open Source* клон *Amazon Dynamo*, также является *AP*-системой. В качестве основного хранилища использует сборку *BerkeleyDB*. Технология, используемая компанией *Linkedln*. Одной из ключевых особенностей системы является высокая доступность операций записи, для чего поддерживается возможность параллельного изменения данных (используется управление конкурентным доступом с помощью многоверсионности (*MVCC*) с алгоритмом векторных часов). Использует консистентное хеширование.
* Имеет высокую вертикальную масштабируемость. Кроме обычного сетевого доступа, *Voldemort* поддерживает *JavaAPI* и различные сетевые протоколы, что сильно экономит трафик и повышает скорость. Данные хранятся как в памяти, так и на диске, сбои питания не нарушат целостности. Сильной стороной является поддержка версионности, то есть каждая единица данных имеет историю версий и изменений. Высокое быстродействие: 10-20 тысяч операций в секунду.

Redis

* *Redis* — сетевое журналируемое хранилище данных типа «ключ-значение» с открытым исходным кодом. *Redis* хранит таблицы в оперативной памяти, что очень удобно для кеширования, также снабжена механизмами снимков и журналирования для обеспечения постоянного хранения. Поддерживает транзакционность и репликацию, а также множество языков программирования, среди которых *C#*, *PHP*, *Ruby* и даже *ActionScript* 3.0.
* По умолчанию, *Redis* не является распределённой системой. Однако можно настроить репликацию пула серверов. Один узел обеспечит высочайший уровень согласованности. Добавление новых узлов в ферму обеспечит увеличение доступности, но усложнит обеспечение согласованности, по причине того, что репликация в *Redis* имеет структуру *Master-Slave*, и все согласованные записи в БД производятся по нисходящей от ведущего сервера к ведомым. Устойчивость к разделению — слабое место *Redis*. Вся логика обработки исключений, связанных с разделением, ложится на приложение.
* В отличие от других хранилищ «ключ-значение» позволяет осуществлять поиск не по конкретному ключу, а по их диапазону (поиск по диапазону чисел или по регулярному выражению). Также позволяет хранить строки, хеш-таблицы, списки и множества, а также имеет базовые операции над этими типами данных.
* Официально не поддерживает *Windows*, однако *Microsoft* разрабатывает экспериментальную версию *Redis* под *Windows*.

Применение хранилищ типа «ключ-значение»

* Хранилища типа «ключ-значение» отлично подходят для таких типов данных, доступ к которым всегда должен осуществляться по уникальному ключу. Например, информация о сессии пользователя веб-приложения всегда имеет *SID* — идентификатор сессии. Да и сам пользователь веб-приложения всегда каким-то образом идентифицируется: будь то *userID*, уникальное имя, поэтому хранение профилей в хранилище «ключ-значение» является обоснованной.

* *Dynamo* используется на *amazon.com* в качестве основного хранилища, то есть большой интернет-магазин — область применения хранилищ «ключ-значение».

Когда не стоит использовать такого рода хранилища:

* Данные плотно связаны между собой. Существуют связи между данными или ключами.

* Необходима транзакционность. Если необходимо произвести несколько операций над несколькими ключами как монолитную транзакцию.

* Необходимо производить запросы по данным. Если необходимо обращаться к пользователю, не по его *ID*, а по *email*.

* Необходимо производить операции над множествами. Мы имеем доступ только к одному ключу. Операции над множествами (такие как суммирование и т.д.) в таком случае придётся выносить в логику приложения.

Колоночные хранилища

* Идея хранить и обрабатывать данные по колонкам (столбцам), зародилась в области бизнес-аналитики для создания высокопроизводительных приложений. Стремительное развитие баз данных этого класса началось после появления *BigTable от Google*. Потом появились *HBase*, *Hypertable* и *Cassandra*.

* Колоночные хранилища (*column-family*, *wide column*) организуют данные в **семейства столбцов**, т.е. наборы строк с непостоянным числом столбцов, ассоциированных с ключом. Аналог семейства столбцов в реляционной модели данных — *таблица*. В отличие от строкового хранения данных у каждой строки не обязательно одинаковый набор столбцов и нет необходимости хранить *NULL* значения для каждого неиспользуемого столбца одной строки

* Если столбец состоит из ещё одного набора столбцов, то он называется **суперстолбец** (*super column*, *wide column*). Набор строк, содержащих суперстолбцы — **семейство суперстолбцов**. Семейства стандартных столбцов и суперстолбцов образуют **пространства ключей**, что аналогично *базе данных* в реляционной модели. СУБД данного типа хранят данные в строках и столбцах, причем масштабируемость достигается посредством разделения между узлами как строк, так и столбцов:

* строки разделяются между узлами с помощью шардинга по первичному ключу;

* для разделения колонок используется группирование столбцов. Каждая из групп колонок обрабатывается как отдельная таблица (для каждой из них шардинг осуществляется отдельно). Колоночные хранилища позволяют использовать скалярные значения полей -

*Cassandra

Apache Cassandra — распределённая СУБД, рассчитанная на создание высокомасштабируемых и надёжных хранилищ огромных массивов данных, представленных в виде хэша. *Cassandra* была разработана в *Facebook*, и используется в этой социальной сети в качестве основного хранилища.

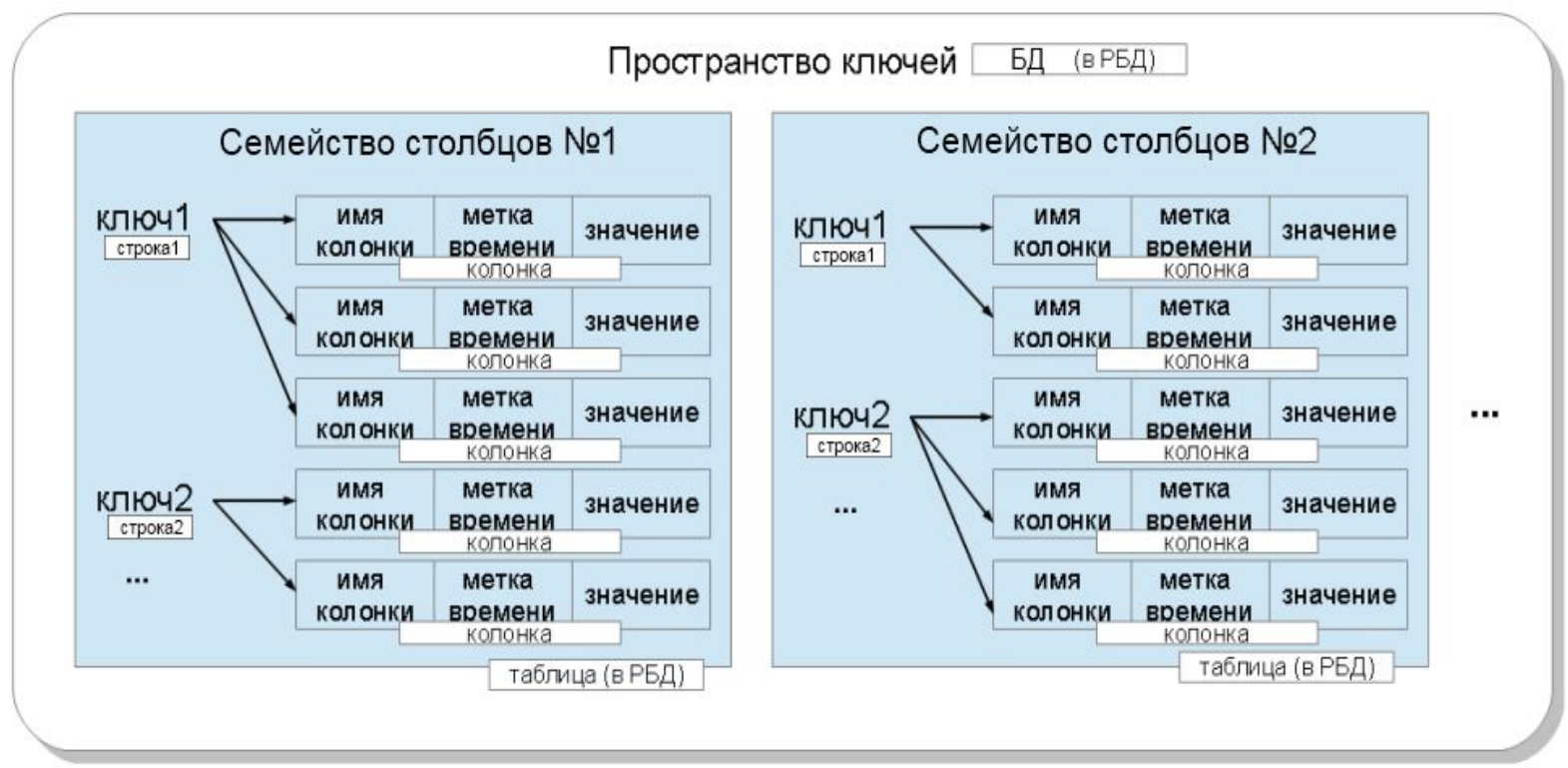


Схема поколоночного хранения данных на примере *Cassandra*

Особенности *Cassandra*

* Столбец в *Cassandra* состоит из пары имя-значение, где имя также является ключом. Каждая такая пара всегда дополняется меткой времени в *UNIX*-формате (*timestamp*) для последующего разрешения конфликтов записи и прочего. Ключ и значение хранятся как массив байтов, имеется понятие типов данных, которые могут создаваться разработчиком, а могут использоваться уже существующие.
* *Cassandra* является распределённой *AP*-системой с *согласованностью в конечном счёте*. Хранилище само заботится о проблемах наличия *единой точки отказа*, отказа серверов и о распределении данных между *узлами кластера*.

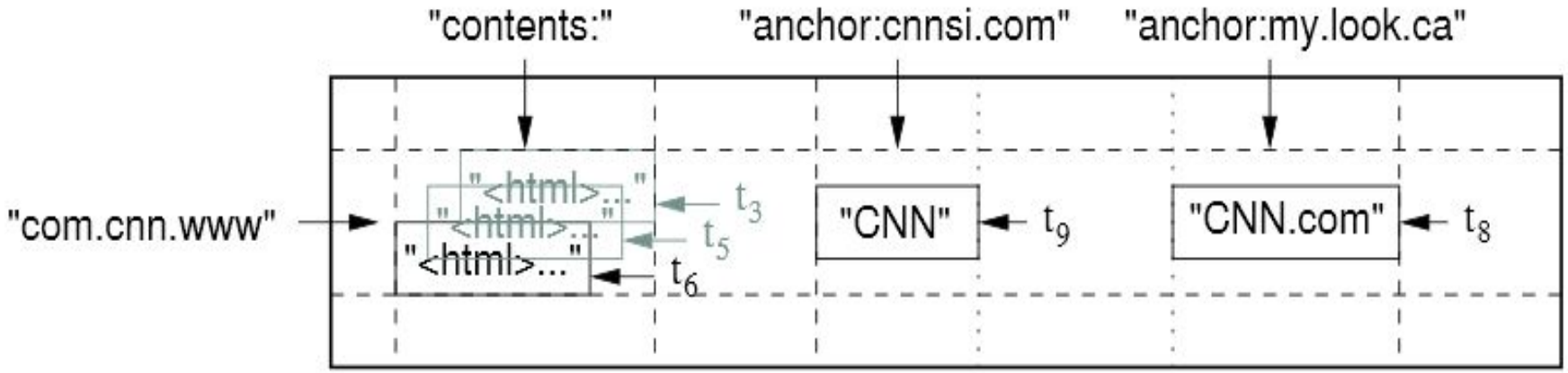
Cassandra поддерживает *транзакционность* на уровне одной записи, то есть для набора колонок с одним ключом. Вот как выполняются четыре требования *ACID*:

* *атомарность* — все колонки в одной записи за одну операцию будут или записаны, или нет;
* *согласованность* — есть возможность использовать запросы со строгой согласованностью взамен доступности, и тем самым выполнить это требование;
* *изолированность* — во время записи колонок одной записи другой пользователь, который читает эту запись, увидит полностью старую версию записи или новую версию после окончания операции, а не часть колонок из одной и часть из второй;
* *долговечность* обеспечивается наличием журнала закрепления, который будет воспроизведён и восстановит узел до нужного состояния в случае какого-либо отказа

Google BigTable

* *Google BigTable* — распределённое хранилище, разработанное *Google* специально для манипулирования огромным объёмом структурированных данных, очень высоко масштабируемое — петабайты данных на тысячах связанных серверов. Многие проекты *Google* работают на *BigTable*, включая веб-индексирование, *Google Earth* и *Google Finance*. *BigTable* — одна из самых ранних *NoSQL* СУБД, оказала огромное влияние и послужила моделью для многих последующих нереляционных хранилищ (например, *Cassandra* и *HBase*).
* *BigTable* — это распределённая разреженная многомерная сортированная таблица, индексирвоанная по ключу строки, ключу столбца и временной метке. Значение в записи представляет собой массив байтов.
* Строковые ключи хранятся в лексикографическом порядке и используются для партиционирования данных. Колоночные ключи разбиваются на группы по префиксу ключа. Временная метка показывает время изменения данных. Данные в БД отсортированы по убыванию значений временных меток, чтобы наиболее свежая информация была доступна в первую очередь. *BigTable* отлично масштабируется и горизонтально, и вертикально, обеспечивает согласованность в конечном счёте.
* *Google BigTable* не является проектом с открытым исходным кодом, и распространяется по модели *Database as a Service* в рамках *Google App Engine*.

BigTable, хранящий веб-страницы в *Google*

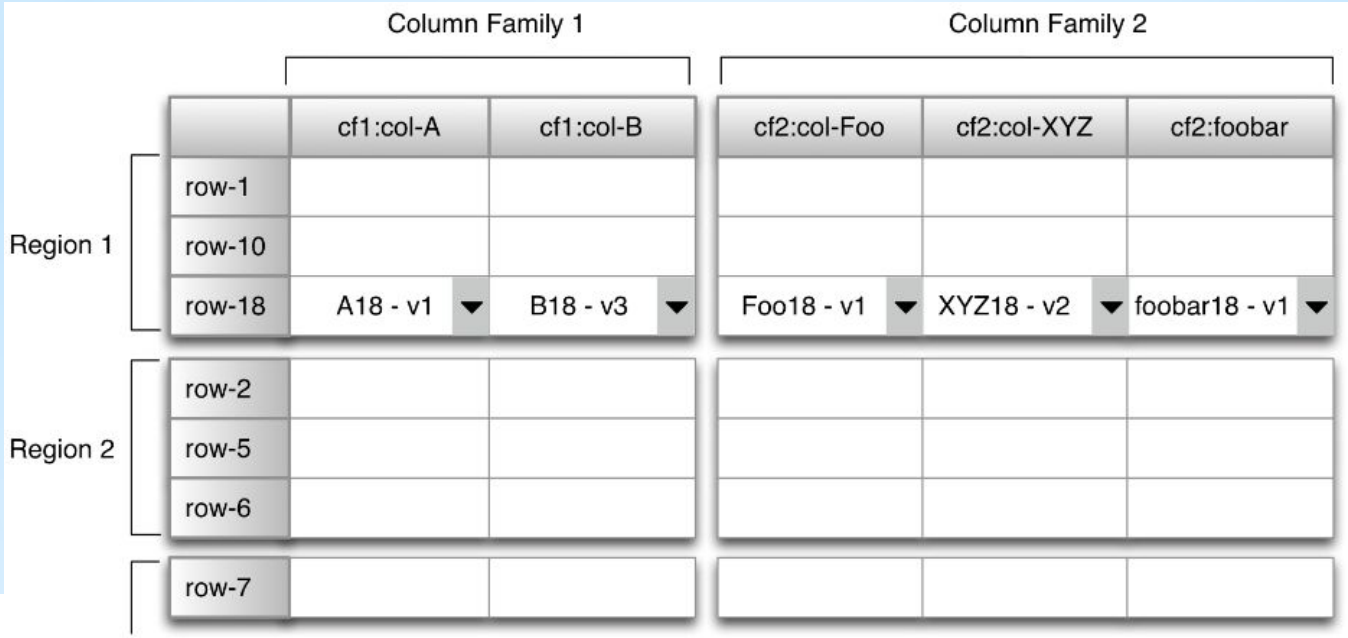


На рисунке изображён пример использования *BigTable* для хранения веб-страниц поиска *Google*: имя строки — реверсивный *URL* адрес страницы, «*contents*» — семейство столбцов, хранящее версии *HTML*-содержимого страницы (с временной меткой t3, t5, t6), «*anchor*» — семейство столбцов, хранящее обратные ссылки (т.е. ссылки на эту страницу).

*HBase

* *Apache HBase* — нереляционная распределённая база данных с открытым исходным кодом; написана на *Java*; является аналогом *Google BigTable*. Поддержка компрессии, операции в памяти и фильтры Блума для каждого базового столбца реализованы *HBase* в соответствии с документацией *BigTable*.
* Таблицы в *HBase* могут служить входом и выходом для работы реализации *MapReduce* в проекте *Hadoop*, и могут быть получены, не только через *Java API*, но и через *API REST*, *Avro* или *Thrift*.
* Является *CP*-системой. Выполняет сильно согласованные запросы на запись и на чтение. Масштабируется горизонтально и вертикально, путём добавления так называемых Регион-Серверов. Обеспечивает автоматический шардинг.

Таблица *HBase*



Таблицы *HBase* состоят из регионов — наборов строк в семействах столбцов. Каждый регион может располагаться на различных узлах кластера (т.е. регион-серверах): таким образом единица масштабируемости *HBase* — регион. Семейства столбцов состоят из неограниченного числа колонок, каждая колонка состоит из неограниченного числа версий. Все значения, кроме названий таблиц — массивы байтов. Доступ к значению производится по такой схеме: (*таблица*, *строка*, *семейство*:*столбец*, *метка времени*) □ *значение*

* Использование колоночных хранилищ

* Реляционные хранилища считают базовой единицей хранения данных строку, и это обеспечивает им хорошую производительность при частой записи данных.
* В то же время, существует множество сценариев, когда гораздо чаще необходимо чтение нескольких колонок у множества строк — обычная модель для большинства веб-приложений. В такой ситуации лучше справляются колоночные СУБД, для которых базовой единицей считается столбец.
* Колоночные хранилища с их способностью хранить любые структуры данных отлично подходят, например, для хранения информации о событиях приложений, такими как состояния приложений и ошибки, возникшие в результате их работы. Все приложения системы обычно имеют различную структуру, и информация о состояниях и ошибках записывается по-разному. Для колоночного хранилища это не проблема — любое приложение может записывать свои состояния с любыми своими атрибутами в базу.
* Используя колоночные хранилища, можно хранить записи в блогах с их тэгами, категориями, ссылками в разных колонках. Комментарии также могут храниться в той же строке, а могут быть вынесены в другое пространство ключей. Также с помощью колоночных хранилищ удобно создавать счётчики посещений страниц.
* Колоночные хранилища не следует использовать, если приложению необходимы *ACID*-транзакции.
* Если сравнивать этот тип хранилищ с реляционными СУБД: РСУБД имеют высокую цену на смену структуры данных по сравнению со сменой запроса, а в колоночных СУБД,

Использование колоночных хранилищ

* Ещё одна интересная возможность — создание временных столбцов (например, у *Cassandra*). Такие столбцы создаются с пометкой *Time To Live* — сколько времени они должны существовать. С их помощью можно давать демо-доступ к части приложения или помещать на некоторое время информацию на сайте.
* Поисковик *Google* использует в качестве основного хранилища, как и многие другие проекты *Google*. *Cassandra* используется в *Facebook* и *Twitter*, а *HBase* — в российской социальной сети ВКонтакте. Таким образом, можно сделать вывод, что колоночные хранилища являются хорошим решением для высоконагруженных интернет-проектов, в которых выборка данных происходит гораздо чаще, нежели запись.
* *HBase* хранилище используется, если в проекте предполагается нагрузка в миллионы или даже в миллиарды строк. Если же количество строк не превышает миллиона, *Apache* советуют использовать обычную РСУБД по причине того, что все данные легко уместятся в одном узле, а остальные даже не будут участвовать, и смысла в масштабировании нет.
* При переходе с реляционной модели на *HBase*, следует убедиться, что Вам не потребуются качества РСУБД, такие как типизированные столбцы, вторичные индексы, транзакции и сложный язык запросов. Кроме того, необходимо убедиться в наличии достаточного количества серверов, так как распределённая файловая система *Hadoop*, на которой построен *HBase*, не показывает себя с лучшей стороны до тех пор, пока в кластере не будет хотя бы пяти серверов.

* Документно-ориентированные хранилища

* Центральным понятием этого подхода является «документ». Он инкапсулирует данные, как правило, в виде одного из широко распространенных форматов: ***XML*** (*eXtended Markup Language*), ***JSON*** (*JavaScript Object Notation*), ***BSON*** (*Binary JSON*), ***YAML*** (*YAML Ain't Markup Language*). Также в этом качестве могут выступать документы в более традиционном смысле - файлы ***PDF*** или ***MS Word***.

* Документно-ориентированные базы данных представляют собой хранилища «ключ-значение», у которых в *значении* хранится документ, то есть *значение* имеет некоторый внутренний формат. Информация содержится не в текстовом виде, а в этом внутреннем формате, позволяющем быстро выполнять операции поиска (запросы *XPath* и *XQuery*) и изменения документов (*XSLT*, *eXtensible Stylesheet Language Transformations*) — т.е. мы имеем возможность работать со значениями, а не только с ключами, в отличие от хранилищ «ключ-значение».

* Документо-ориентированные СУБД обычно поддерживают вторичные индексы, а также вложенные документы и списки. Такие базы данных больше, чем хранилища «ключ-значение», напоминают РСУБД, но имеют ряд важных отличий: данные не привязаны к жесткой схеме, а набор полей (ключей, секций) у документов может различаться.

* Так же, как и другие *NoSQL*-системы, документно-ориентированные базы данных обычно не удовлетворяют *ACID*-требованиям.

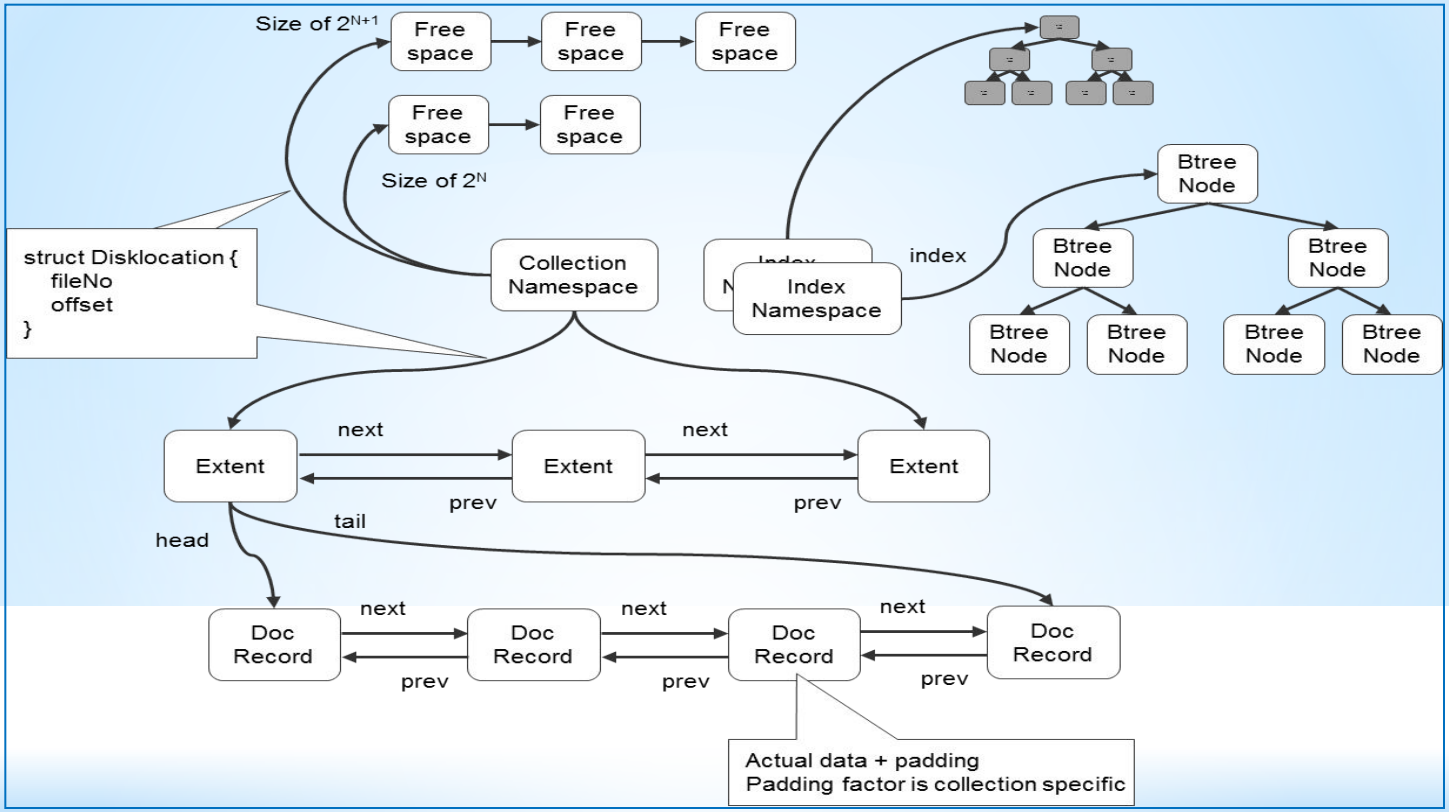
MongoDB

* СУБД *MongoDB* управляет наборами *JSON*-подобных документов, хранимых в двоичном виде в формате *BSON*. *Инстанс MongoDB* может содержать несколько *баз данных*, каждая из которых может содержать *коллекции* (аналог в РСУБД — *таблицы*).
* Имеется множество драйверов для работы с *MongoDB* с помощью различных языков программирования: *C#*, *C++*, *PHP*, *Java*, *JavaScript*, *Node.js*, *Python*, *Perl*, *Ruby*, и др.
* Хранение и поиск файлов в *MongoDB* происходит благодаря вызовам протокола *GridFS*.
* Среди прочих отличий от традиционных реляционных СУБД:
* Отсутствует оператор «*join*». Обычно данные могут быть денормализованы и на разработчиков ложится дополнительная нагрузка по обеспечению непротиворечивости данных.
* Нет такого понятия, как «транзакция».
* Отсутствует понятие «изоляции». Любые данные, которые считываются одним клиентом, могут параллельно изменяться другим клиентом.

Основные возможности *MongoDB*

* Достаточно гибкий язык для формирования запросов
* Динамические запросы
* Полная поддержка индексов
* Профилирование запросов
* Быстрые обновления «на месте»
* Эффективное хранение двоичных данных больших объёмов (фото и видео)
* Журналирование операций, модифицирующих данные в БД
* Поддержка отказоустойчивости и масштабируемости: асинхронная репликация, набор реплик и шардинг
* Может работать в соответствии с парадигмой *MapReduce*
* Полнотекстовый поиск, в том числе на русском языке, с поддержкой морфологии
* *MongoDB* использует файл карты памяти, который непосредственно отображает дисковый файл данных в байтовый массив в памяти, где логика доступа к данным реализована с использованием арифметики указателей. Каждая коллекция документов хранится в одном файле пространства имен (который содержит информацию

*Структура данных *MongoDB*



Структура данных широко использует двунаправленный связанный список. Каждая коллекция данных организована в связанном списке степеней, каждая из которых представляет непрерывное дисковое пространство. Каждая степень указывает на голову/хвост другого связанного списка документов. Каждый документ содержит связанный список к другим документам, к тому же актуальные данные закодированы в формате BSON.

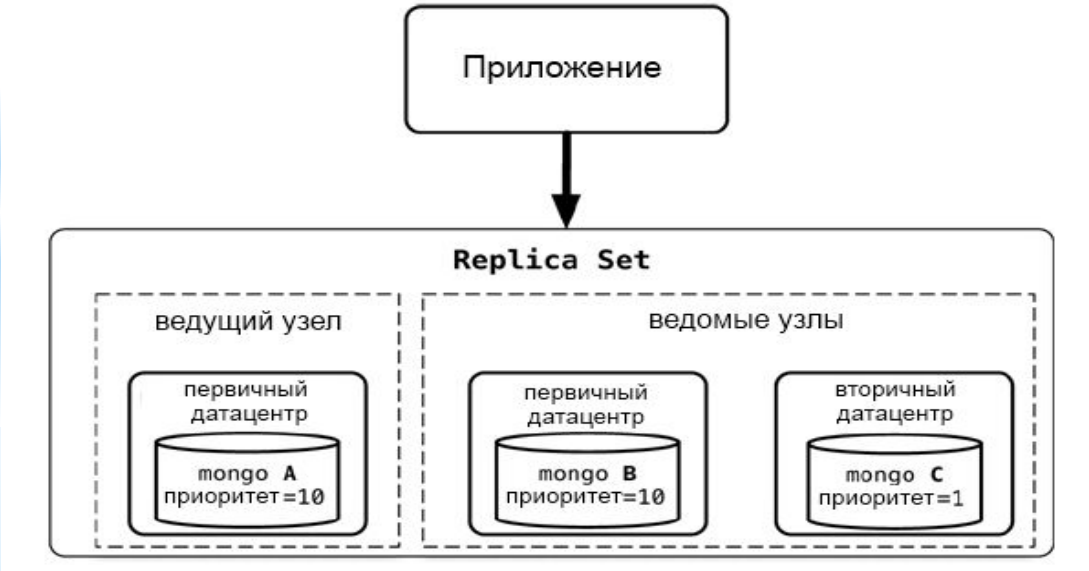
Модификация данных

* Модификация данных происходит на месте. В случае если модификация увеличивает размер записи вне его первоначально выделенного места, вся запись будет перемещена в большую область с несколькими дополнительными байтами. Дополнительные байты используются в качестве буфера роста так, чтобы будущее расширение не требовало повторного перемещения данных. Количество дополнения динамически корректируется под коллекцию на основе ее статистики модификации. С другой стороны, место, занятое исходным документом, будет свободно. Это отслеживается в списке свободного места различных размеров. Таким образом, можно предположить, что в течение долгого времени будут образовываться дыры, поскольку объекты создаются, удаляются или изменяются, эта фрагментация повредит производительности, так как меньше данных будут читаться/записываться на один дисковый ввод-вывод. Поэтому, мы должны периодически выполнять команду «*compact*», которая копирует данные в непрерывное пространство. Эта "компактная" работа является исключительной операцией и должна быть сделана оффлайн.

* Индекс в *MongoDB* реализован как *B*-дерево. Каждый узел *B*-дерева содержит определенное число ключей (в этом узле), а также указатели на левые дочерние узлы *B*-дерева каждого ключа.

Репликация в *MongoDB*

* *MongoDB* поддерживает репликацию по схеме «Ведущий-Ведомый», в основном для повышения отказоустойчивости (и, как следствие, доступности). Основным понятием в этой репликации является *Replica Set* — кластер инстансов *MongoDB*, среди которых происходит автоматическая обработка отказов и выбор нового ведущего, если старый перестал отвечать. Согласованность внутри *replica set* регулируется с помощью параметра для запроса на запись, который указывает, сколько ведомых серверов должны выполнить запись, прежде чем продолжить. Кроме того, для увеличения производительности, механизм позволяет производить чтение с ведомых узлов, что обеспечивается добавлением *JavaScript*-метода *.slaveOk*() к объекту соединения, к базе данных, коллекции или к каждой отдельной операции.
* Добавление серверов в *replica set* происходит прозрачно и на лету, без необходимости приостановки работы кластера.



*CouchDB

* *CouchDB* — *NoSQL*-СУБД, разработанная *Apache*. Позволяет использовать в качестве полей документа скалярные значения (числа, строки), а также списки и вложенные документы. Типизация элементов данных, то есть сопоставление отдельным полям документов типов *INTEGER*, *DATE* и пр., не поддерживается — вместо этого пользователь может написать функцию-валидатор, обычно на языке *JavaScript* или *ErLang* (на котором и был написан *CouchDB*), которые затем хранятся на сервере в текстовом виде.
* Возможно использование вторичных индексов, которые должны создаваться явным образом. Индексы реализованы с помощью *B*-деревьев, что позволяет упорядочивать результаты выборки, а также осуществлять выборку по диапазону значений.
* *CouchDB* — одна из немногих *NoSQL* СУБД, у которых реализован полноценный графический веб-интерфейс пользователя — *Futon*.

Кроме того, к серверу предоставляется доступ посредством протокола *REST*, то есть можно производить к серверу в следующем виде:

* Если мы хотим получить информацию — отправляем *GET* запрос.
* Если надо создать документ — *POST*
* Необходимо что-то изменить — *PUT*
* *COPY* для копирования

Особенности *CouchDB*

- Запросы могут выполняться параллельно к множеству узлов - для этого используется *MapReduce*. Также *CouchDB* позволяет управлять параллельным доступом к БД с помощью паттерна *MVCC — Multi-Version Concurrency Control*.
- Все документы в *CouchDB* определяются версией. Если документ изменяется, создаётся его полная новая версия и записывается поверх старой (в то время как старая ещё хранится). Такой механизм спасает от блокировок на запись. Если обычно СУБД блокирует сущность, изменяемую кем-то в данный момент, и ставит все попытки чтения в очередь на ожидание завершения изменений, то *CouchDB* на любой запрос чтения возвращает последнюю доступную версию документа. Даже если в момент запроса документ кем-то изменяется, то нам это не мешает взять старую его версию. Как только изменение завершится, и запишется новая версия поверх старой, по запросу на чтение будет доступна уже она.
- В *CouchDB* документ представляет собой *JSON* объект, который, кроме заданных пользователем полей, обязательно содержит специальные поля *_id* — идентификатор элемента и *_rev* — ссылка на версию элемента. Также документ *CouchDB* может хранить файлы, которые складываются в массив под названием *_attachments*.

Масштабируемость

- Масштабируемость достигается посредством так называемой *инкрементальной репликации* — изменения документа периодически копируются между серверами. В случае если один документ был изменён сразу в двух базах, «выигрывает» более поздняя версия, однако «проигравшая» тоже записывается и становится предыдущей версией. Шардинг не используется. Таким образом, *CouchDB* является *AP*-системой и обеспечивает согласованность в конечном счёте.
- Несмотря на то, что *CouchDB* изначально предназначался для работы в операционной системе *Linux*, уже разработаны варианты этой системы для операционных систем *Microsoft Windows* и *Mac OS*.

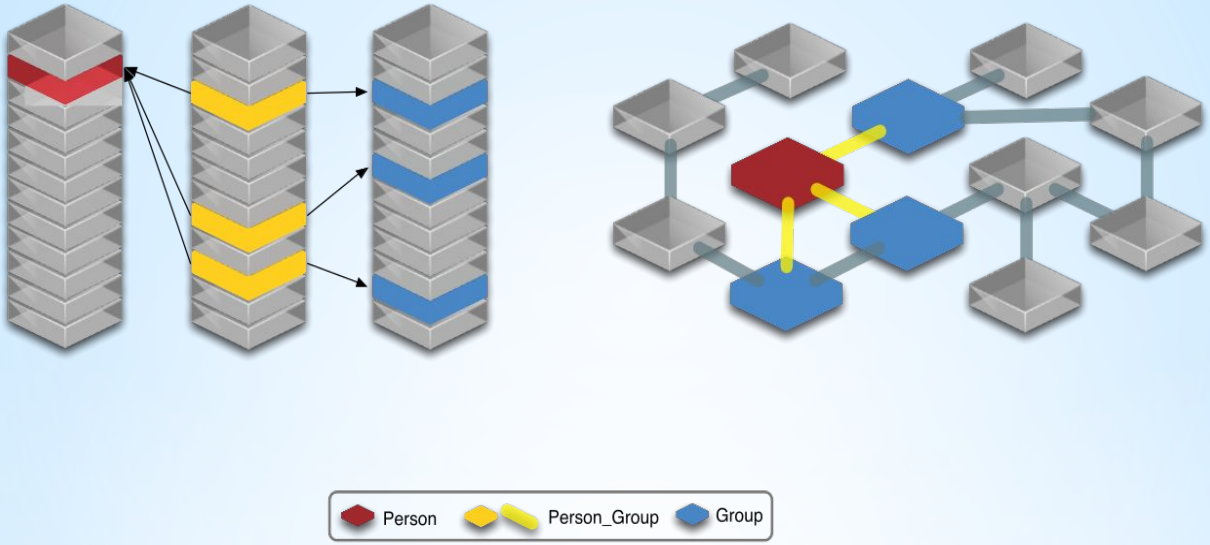
Использование документно-ориентированные хранилищ

* Документно-ориентированные хранилища представляют собой очень удобную прослойку между схемами хранилищ «ключ-значение» и реляционными СУБД. Они подходят для хранения документов, у которых просматривается определённая структура, но всё же довольно сильно варьируется.
* Так как такого рода хранилища не имеют строго определённой структуры и обычно понимают *JSON* формат, они хорошо работают в *CMS* (*Content Management System*) — системах управления контентом сайта или в публичных веб-сайтах, храня регистрацию, профили и комментарии пользователей.
* Так же как и колоночные, документно-ориентированные СУБД подходят для журнала событий. Кластер серверов *MongoDB* может являться основным хранилищем, содержащим в себе журналы событий всех приложений.
* Так как документы в хранилище могут быть изменены «на лету», такие СУБД подходят для систем аналитики реального времени. Кроме того, аналитика обычно требует серьёзных расчётов, и вполне разумно хранить кэш — результат предыдущего анализа в *XML* документе, например, в *HTML*-странице.
* Документно-ориентированные СУБД не стоит использовать, если необходимы атомарные транзакции между несколькими документами. Не подходит, если необходимы частые незапланированные запросы. Необходимость таких запросов обычно объясняется частым изменением структуры запрашиваемых данных. А так как результаты запросов к документно-ориентированным СУБД необходимо интерпретировать программой, то такая частая перекомпиляция или просто перекодирование структуры, может нарушить все плюсы данного подхода.

Хранилища графов

* Базы данных на основе графов позволяют хранить *сущности* и *связи* между ними. Сущности в рамках таких хранилищ являются *узлами*, они могут иметь свойства. Связи организуют сущности и представляются как *рёбра* графа, и они тоже могут иметь свойства, кроме того они всегда имеют направление.
* Преимуществом такого построения хранилища перед другими *NoSQL* системами (такими как «ключ-значение» и колоночные) является тот факт, что нам не нужно менять структуру данных при желании изменить запрос, т.к. мы можем выполнять обход графа любым доступным способом.
* Реляционная модель тоже может содержать граф, используя связки и внешние ключи, однако добавление новой связи или изменение существующей обычно также означает серьёзные изменения в структуре, но для графа не является проблемой.
* Обход связей, или соединений (*join*) выполняется хранилищем графов гораздо быстрее, нежели реляционным, потому что связи в графе являются постоянными, а не высчитываемыми во время запроса. Для хранилища графов нет необходимости хранить связи в отдельных сущностях, связи обрабатываются совершенно по-другому.

* Сравнение структуры связей в реляционной и в графовой модели данных



Ещё одним преимуществом графовых БД перед реляционными является то, что они сразу ориентированы на большое количество связей между данными. Скажем, если нам требуется производить такого рода запросы: выдать тех, кто имеет отношение к другим, которые имеют отношение к третьим и т.д., то с каждым добавлением такого «рукопожатия» в реляционном запросе будет добавляться вложенный подзапрос, и, в конечном счете, всё это приведёт к тому, что запрос станет неподъёмным. Для графовых хранилищ такие запросы — обычное дело.

Neo4j — графовая база данных

* *Neo4j* — графовая база данных с открытым исходным кодом, поддерживаемая компанией *Neo Technology*. Neo4j хранит данные в узлах, связанных направленными типизированными отношениями со свойствами как для узлов, так и для отношений.

Преимуществами Neo4j являются:

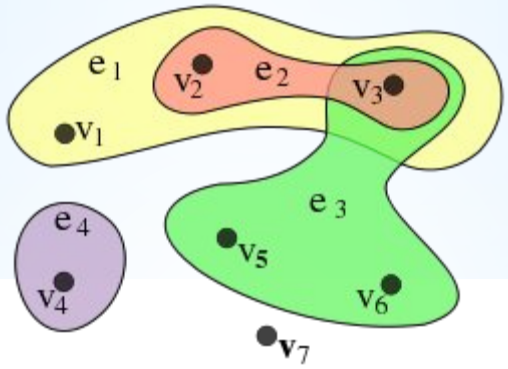
* интуитивно-понятная модель представления данных при помощи графа;
* полная поддержка *ACID* транзакций. Перед любым изменением узлов необходимо начать транзакцию;
* высочайший уровень масштабируемости — до нескольких миллиардов узлов, связей и свойств;
* возможность репликации по схеме «ведущий-ведомый». Ведущий выбирается автоматически с помощью *Apache ZooKeeper*;
* человеко-читаемый язык запросов для графов *Cypher*.
* быстрый фреймворк для запросов по обходу графа — нахождение путей между двумя узлами, нахождение кратчайшего пути по алгоритму Дейкстры и т.д.;
* свойства узлов и отношений могут индексироваться с помощью специального сервиса *Lucene*. Также существует возможность создания полнотекстового индекса;
* встраиваемость в приложения;
* простой доступ к данным с помощью *API REST* или объектно-ориентированного *Java API*;
* существуют драйверы для множества языков программирования, в том числе для *.NET*;
* существует несколько графических интерфейсов: *Neo4j Webadmin*, *Neoclipse*, *Gephi*

Недостатки:

* не поддерживает операцию *join*;
* данные разрежены;

*HyperGraphDB

* *HyperGraphDB* — это расширяемая портативная распределённая встраиваемая система хранения данных с открытым исходным кодом. Это графовая база данных, спроектированная специально для систем искусственного интеллекта и семантических веб-проектов, которая также может использоваться как встраиваемая объектно-ориентированная БД. Изначально создана для описывания гиперграфов — графов, в которых одно ребро может соединять любое количество узлов. На низшем уровне основана на *BerkeleyDB*.



Гиперграф

* Базовый элемент данных — атом, который типизирован, имеет значение и может указывать на любое количество других атомов. Типы данных — тоже атомы, с определенными ролями. Отсутствует ограничение на дисковое пространство, но каждое конкретное значение (атом) может быть не больше 2 ГБ (ограничение *BerkeleyDB*).
* Поддерживает *ACID* транзакции, *MVCC* алгоритм для параллелизма, репликацию, автоматический шардинг, индексирование, *Java API* для обхода графа. *Согласованность* обеспечивается механизмом *STM — Software Transactional Memory*, контролирующем доступ к распределённой памяти (*shared memory*) по тому же принципу, что и *ACID* в транзакционных моделях.

Использование хранилищ графов

* Хранилища графов подходят для хранения плотно связанных данных, когда основной упор делается не на сущности, а именно на связи между ними. Особенно, если необходимы глубокие запросы к таким связям (прохождение нескольких «рукопожатий»). Очевидным примером такой системы служат социальные сети. Такие социальные графы не обязательно должны иметь связь только одного типа — «друг»; например, они могут представлять сотрудников, их знания, где и с кем они работали, над какими проектами.
* Маршрутизация, рассылка, и геолокационные сервисы могут быть описаны с помощью графов. Свойства отношений между узлами в таком случае могут содержать расстояние, вес (или приоритет) ребра. С помощью таких систем можно высчитывать кратчайший путь между любыми узлами для эффективной доставки чего-либо от одного к другому.
* Системы рекомендаций, например «*like*» в *Facebook* или «вместе с этим товаром покупают» в интернет-магазинах, тоже отлично подходят как задачи для графовых баз данных.
* Хранилища графов не являются хорошим решением в тех случаях, когда связей между данными не так много или они не так значительны — в таком случае просто теряются все плюсы такого подхода. Кроме того, существуют ситуации, когда требуется изменение всего множества сущностей (или его большого подмножества), с которыми графовые БД плохо справляются по той причине, что изменение свойств на всех узлах сразу сильно понижает производительность такой БД, требуя огромных ресурсов.

Достоинства и недостатки NoSQL

Можно сделать вывод о том, что *NoSQL* не является «панацеей» — у нереляционного подхода, как и у реляционного, можно выделить как сильные, положительные стороны, так и отрицательные.

Несомненными плюсами NoSQL являются:

- высокая скорость работы;
- превосходная масштабируемость (вертикальная и/или горизонтальная);
- нетипизированность атрибутов и наборов атрибутов;
- отсутствие языка запросов.

К минусам же следует отнести:

- отсутствие языка запросов (это не только плюс, но и минус);
- сложность проектирования хранилищ и систем, на них построенных;
- ресурсоёмкость в плане памяти, как физической (из-за избыточности), так и оперативной (из-за хранения некоторыми СУБД таблиц в оперативной памяти);

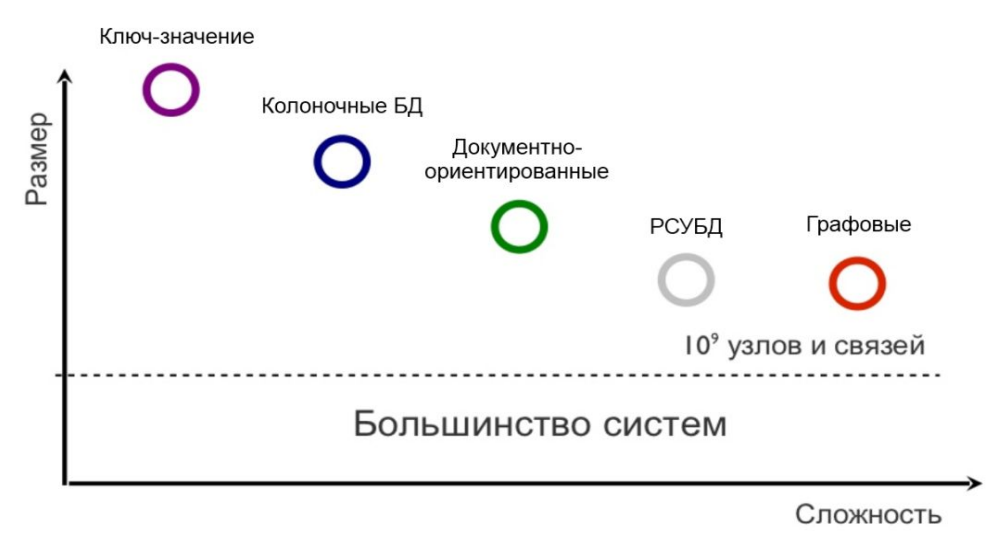
По этим причинам нужно чётко понимать, стоит использовать NoSQL в каком-либо проекте или нет.

SQL и *NoSQL* движутся навстречу друг другу, а со временем могут слиться в единый подход *SQL+NoSQL* к разработке СУБД.

*NoSQL или SQL?

Возникает вопрос: что лучше использовать — *NoSQL* или *SQL*?

- Широко распространенная и хорошо изученная реляционная СУБД дает массу дополнительных преимуществ: интеграция с уже существующими решениями на базе реляционных СУБД, отказоустойчивость, наличие более богатого инструментария и т. д.
- *NoSQL* же обеспечивает высочайшую производительность при обработке огромного количества данных.
- При объединении *SQL* и *NoSQL* в единое решение из-за многообразия используемых методов хранения информации СУБД может превратиться в огромный черный ящик со сложными и разветвленными настройками, что сильно затруднит работу с ней. Похожая ситуация уже происходила с некоторыми языками программирования (например, Алгол-68, *PL*/1), когда их семантика оказалась сильно перегружена и вместо удобного инструмента разработчики получили плохо управляемого «монстра». Кроме того, если будет разработан стандарт на системы *SQL+NoSQL*, то он может оказаться слишком сложным и перегруженным.



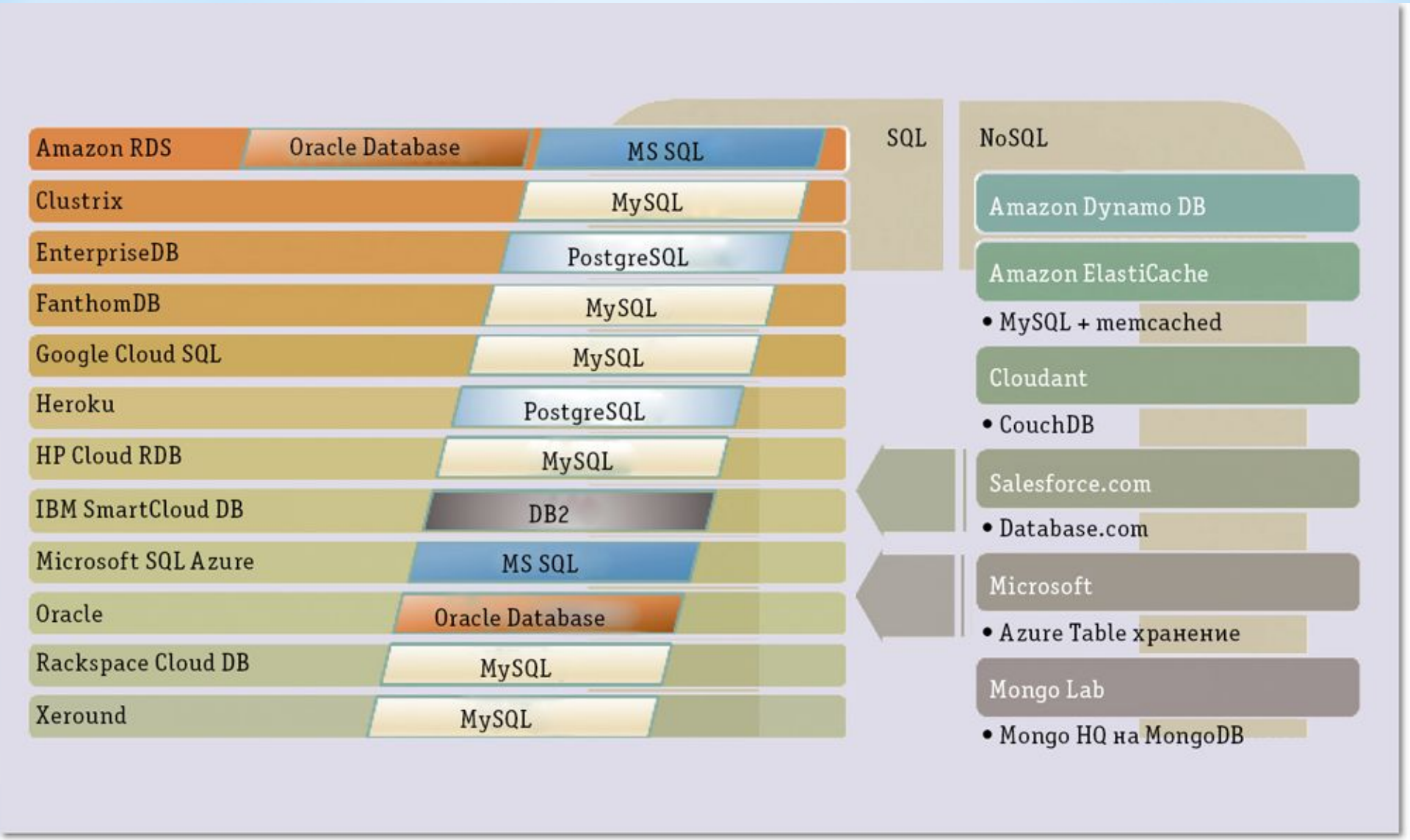
*Выбор решения

*Можно предложить подход при выборе типа хранилища для конкретного проекта. На рисунке показано относительное положение различных типов хранилищ. Выбор хранилища зависит от конкретного разрабатываемого приложения: насколько сильно связаны сущности приложения, насколько сложны эти связи показывает ось *X*; ось *Y* показывает предполагаемый размер хранилища, т.е. количество записей в БД. Таким образом, при простейшей структуре приложения, но огромных количествах записей лучше всего использовать хранилища типа «ключ-значение». Если количество связей в БД велико, требуется транзакционность и т.д., а количество записей не так уж значимо, то обычная РСУБД, такая как *Microsoft SQL Server* отлично подойдёт в любой ситуации. Но если количество отношений между сущностями очень велико и даже превышает количество сущностей, а сами эти отношения играют ключевую роль в системе, то идеальным решением будет графовое хранилище.

*Hadoop

Hadoop— проект фонда Apache Software FoundationHadoop— проект фонда Apache Software Foundation, свободно распространяемыйHadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилитHadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотекHadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворкHadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерахHadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов. Используется для реализации поисковых и контекстных механизмов многих высоконагруженных веб-сайтов, в том числе, для Yahoo!Hadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов. Используется для реализации поисковых и контекстных механизмов многих высоконагруженных веб-сайтов, в том числе, для Yahoo! и FacebookHadoop— проект фонда Apache Software Foundation, свободно распространяемый набор утилит, библиотек и фреймворк для разработки и выполнения распределённых программ, работающих на кластерах из сотен и тысяч узлов. Используется для реализации поисковых и контекстных механизмов

* SQL и NoSQL


SQL
NoSQL
Amazon RDS
Oracle Database
MS SQL
Clustrix
MySQL
EnterpriseDB
PostgreSQL
FanthomDB
MySQL
Google Cloud SQL
MySQL
Heroku
PostgreSQL
HP Cloud RDB
MySQL
IBM SmartCloud DB
DB2
Microsoft SQL Azure
MS SQL
Oracle
Oracle Database
Rackspace Cloud DB
MySQL
Xeround
MySQL
Amazon Dynamo DB
Amazon ElastiCache
• MySQL + memcached
Cloudant
• CouchDB
Salesforce.com
• Database.com
Microsoft
• Azure Table хранение
Mongo Lab
• Mongo HQ на MongoDB

Классификация облачных инфраструктур

*Облачные СУБД

СУБД, предоставляемая по подписке со всеми подобающими харак теристиками — DBaaS (database as a service) облачную технологию в рамках платформенной модели обслуживания.

Облачные СУБД стали одним из магистральных направлений в от расли, а с ростом их коммерческой значимости они становятся драй вером в продвижении к решению таких общих и важных для всей индустрии ИТ вопросов, как построение горизонтально-масштаби руемых транзакционных СУБД, конвергенция реляционных баз дан ных и движения NoSQL, внедрение технологий Больших Данных в СУБД. Растущий в последние годы интерес к Data Intensive Computing в контексте высокопроизводительных вычислений спо собствует развитию направления, ведь идеальный массовый DBaaS — это решение такого «суперкласса», обеспечивающее эф фективное распределенное управление данными в гигантских мас штабах, и многие технологические решения в рамках этих направле ний будут общими.

Требования к облачным СУБД

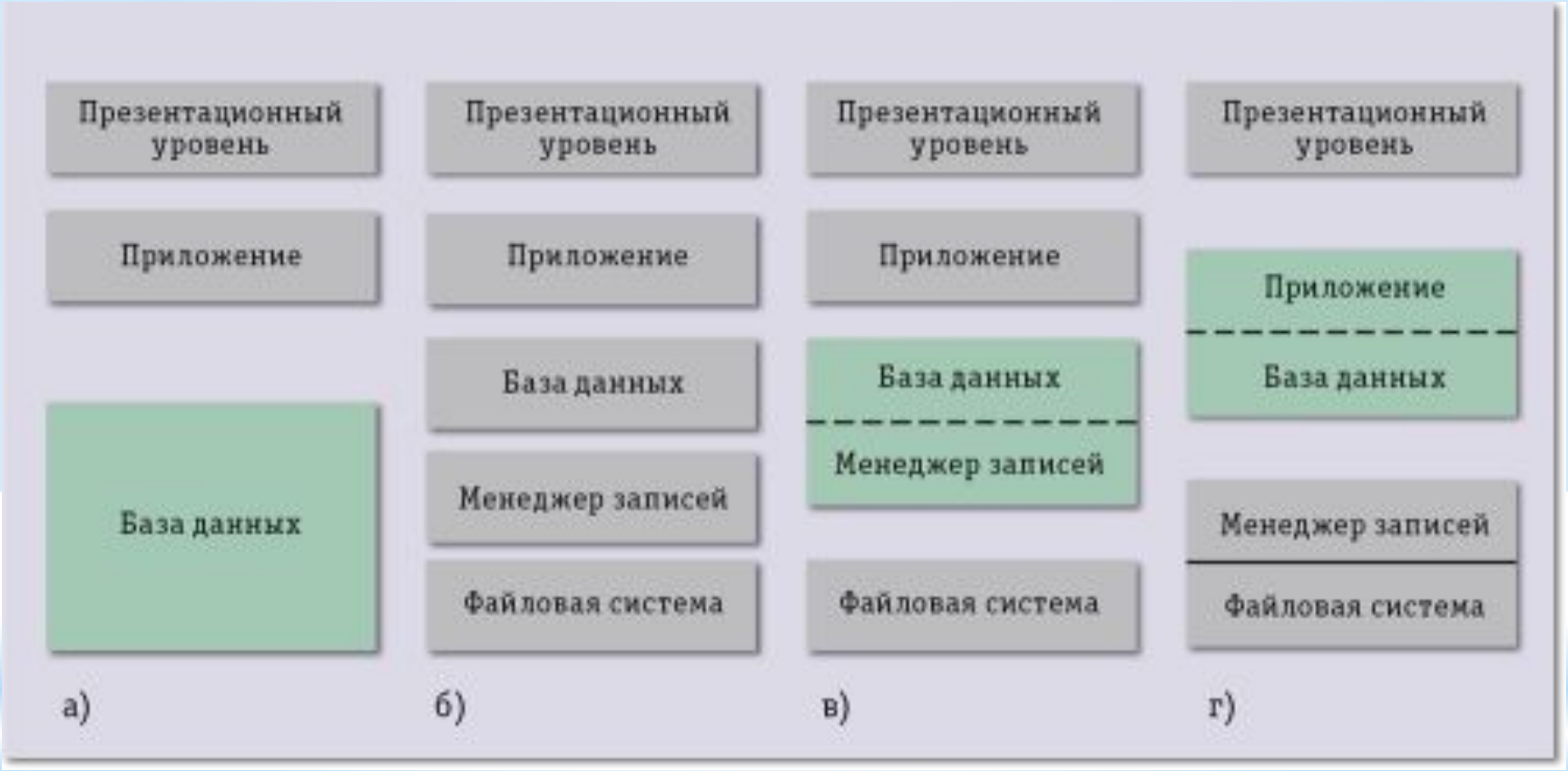
Модель аренды ресурсов с оплатой по мере использования, характерная для облаков, дает разработчикам системных архитектур новые возможности оптимизации затрат, но для получения экономии управление данными в облаке должно отвечать ряду новых требований.

- *Эластичность.* Облачные СУБД должны легко масштабироваться вверх и вниз в зависи мости от рабочей задачи. Таким образом можно избежать избыточности ресурсов и сни зить операционные расходы. Лучше всего подойдет горизонтальное масштабирование, поскольку подключать или отключать дополнительные машины проще, чем модернизи ровать отдельную систему, а кроме того, такое масштабирование теоретически не имеет границ.

- *Гибкость и простота управления.* Процедура развертывания новых приложений, поль зующихся облачной СУБД, должна быть простой. Данное требование подразумевает применение адаптируемых моделей данных и схем.

- *Отказоустойчивость.* Для облачных СУБД необходима автоматическая обработка ава рийных ситуаций, так как сбои в крупномасштабных развертываниях скорее норма, чем исключение.

*Новые архитектуры баз данных



Многозвенные архитектуры: а — классическая трехзвенная, б — пятиз венная, в — четырехзвенная, г — трехзвенная архитектура с хранилищем «ключ-значение». Сплошными линиями обозначены раздельные компонен ты, пунктирные показывают, что компоненты объединены друг с другом

*Новые уровни данных

Традиционная трехзвенная архитектура была преобразована в пятиуровневую: презентационный уровень, приложение, база данных, менеджер записей и фай ловая система. В новых системах эти уровни (слои) по-разному комбинируют ся; каждый слой представляет собой сервис с возможностью независимого масштабирования и управления для отказоустойчивости. Появились два новых уровня, а один был переработан.

Файловый уровень

Этот уровень обеспечивает доступ к файлам и лежит внизу стека. Интерфейс файловой системы предоставляет соответствующие функции записи/считыва ния и обычно оптимизирован для больших файлов. Наиболее известные облач ные примеры файлового уровня: Google File System и Amazon Simple Storage Service. Обычно файлы семантически организованы в иерархическую структу ру — имена файлов в пределах своего локального контекста уникальны.

*Уровень менеджера записей

Менеджер записей расширяет функциональность файловой системы, предо ставляя более мелкодисперсный доступ к данным. В простейшем случае мене джер записей поддерживает только операции чтения и записи одиночных зна чений для каждого ключа. Системы, предоставляющие этот простой интер фейс, стали популярными вместе с развитием движения NoSQL (обычно их называют хранилищами «ключ-значение»). Но многие системы этой катего рии начали предоставлять более развитую функциональность, в том числе ре ляционные модели и возможность запрашивать диапазоны ключей с сортиров кой. В результате получились уже не просто хранилища «ключ-значение», а системы «менеджеры записей», поскольку они реализуют функциональность, похожую на аналогичный компонент СУБД. Менеджер записей может упоря дочить элементы данных в файловой системе так, чтобы ускорять выполнение взаимосвязанных запросов. К этому же уровню можно отнести такие механиз мы оптимизации доступа, как индексы на основе B-деревьев, поскольку они предоставляют тот же интерфейс, что и данные, индексируемые ими. На этом уровне менеджер записей может реализовывать версионный контроль — хра нить по нескольку версий одной и той же записи.

Уровень баз данных

Этот уровень находится поверх менеджера записей, отвечая за об работку запросов и транзакций, а также за обслуживание пред ставлений и индексов. В зависимости от системной модели запро сов, базе приходится выполнять разный объем работы — от ком пиляции SQL-подобных запросов в физические планы выполне ния для менеджера записей до простого принятия решения о том, к каким серверам обращаться для извлечения данных. В систе мах, поддерживающих аналитические запросы (таких как Hadoop MapReduce), база планирует выполнение элементов работ (зада ний) из параллельных запросов на доступных серверах. Большин ство существующих менеджеров записей поддерживают только атомарный доступ к индивидуальным записям, а база дополни тельно может выполнять функции транзакционной системы с со ответствующими гарантиями. Кроме того, на уровне базы данных автоматически создаются и обновляются вторичные индексы.

*Пятизвенная архитектура

В такой архитектуре каждый уровень представляет собой отдельный сервис. Самый известный пример — это стек Google, состоящий из GFS (файловый уровень), BigTable (уровень менеджера записей) и Megastore (уровень баз дан ных). Наверху же находятся уровни приложений и презентационный. Такая се парация обеспечивает высокую модульность, позволяющую разработчикам легко использовать отдельные элементы стека. Например, при аналитической обработке данных с помощью MapReduce можно пользоваться только файло вым уровнем. BigTable, работая поверх файловой системы, предоставляет функциональность простого менеджера записей: доступ к ключам-значениям («строкам» в терминологии Google) с атомарными однострочными операция ми, применяемыми для работы с веб-индексом Google. Уровень Megastore до бавляет к этому поддержку транзакций и самоподдерживаемые индексы. Хотя у такого расслоения есть много преимуществ, оно создает дополнительный се тевой трафик между уровнями: например, операция соединения требует, чтобы данные доставлялись с файлового уровня через диспетчер записей на уровень базы данных. Возможно также снижение эффективности из-за дубли рования функциональности на разных уровнях.

Четырехзвенная разделенная архитектура с файловым уровнем

Один из способов уменьшить трафик между уровнями — объединить базу дан ных и диспетчер записей в один уровень. Внизу, в роли надежной сетевой си стемы хранения данных (Networked Attached Storage, NAS) действует файло вый сервис. Подобно классической трехзвенной архитектуре, такое сочетание позволяет разработчикам пользоваться стандартными СУБД, такими как MySQL, однако за достижение высокой отказоустойчивости в этом случае отве чает файловый уровень, а не база данных. Примеры — Amazon Relational Database Service, Amazon Elastic Compute Cloud (EC2) в сочетании с Amazon Elastic Block Service (EBS), используемым как NAS. Последний позволяет раз работчикам установить СУБД по выбору — можно достигнуть высокой устой чивости, просто сохраняя все журналы операций базы на томе EBS, который можно будет смонтировать на другом экземпляре EC2, если первоначальный даст сбой. К сожалению, такая архитектура наследует ограничения по масшта бируемости от классической трехзвенной, так как обычно использует традици онную монолитную СУБД. Кроме того, трудно согласовать различные протоко лы консистентности, применяемые внутри файлового уровня и объединенного уровня базы данных и диспетчера записи.

*Интеграция SQL и NoSQL

Одно приложение — много СУБД

SQL и NoSQL во многом можно противопоставить друг другу, но уже обозначилась тенденция не только к сближению этих миров, но и к осознанию того, что нельзя ограничиться чем-то одним для решения всех возникающих задач, — современные приложения становятся все более сложными, и успех реализации часто зависит от осознанного выбора инструментария. В условиях отсутствия универсального решения SQL и NoSQL начинают работать совместно, позволяя добиваться более высоких результатов за счет использования преимуществ обеих технологий.

*Big Data definitions

Простое определение: Большие Данные те, что слишком **велики** и **сложны**, чтобы их можно было эффективно запомнить, передать и проанализировать **стандартными средствами доступных баз данных и иных имеющихся систем хранения, передачи и обработки.**

Кроме объема, следует учитывать и другие их характеристики. Еще в 2001 году Мета Групп ввела в качестве определяющих характеристик для больших данных так называемые ***«три V»*:**

- **объём** (*volume*, в смысле величины физического объёма),
- **скорость** (*velocity* в смыслах как скорости прироста, так и необходимости высокоскоростной обработки и получения результатов),
- **многообразие** (*variety*, в смысле возможности одновременной обработки различных типов структурированных и неструктурированных данных

Однако, когда **общий поток данных растет экспоненциально, удваиваясь каждый год,** за счет революционных технологических изменений, к 2014 году даже эту "3V" модель предлагают расширить, добавляя новые и новые «V»,

- **Validity** (обоснованность, применимость),
- **Veracity** (достоверность),
- **Value** (ценность, полезность),
- **Visibility** (обозримость, способность к визуализации) и т. д.

*How useful is Big Data ?

Большие должны быть доступны для поисковых систем, проанализированы в центрах бизнеса, производства, медицины, правоохранения, обороны, науки и просто индивидумов, которые их могут затребовать.

Что это может дать? – Упростить документооборот, но главное - прогноз!

Пример с вирусом H1N1: Google в 2009 г. точно предсказал распространение гриппа в разных штатах США, проанализировав 3 млрд запросов на лекарства от гриппа на их корреляцию с распространением гриппа во времени и пространстве.

Большие данные помогут решению насущных глобальных проблем, таких как борьба с изменением климата, искоренение болезней, а также содействие эффективному управлению и экономическому развитию. Однако для этого придется пересмотреть традиционные представления об управлении, принятии решений, человеческих ресурсах и образовании.

Как это делается?

http://www.livelib.ru/book/1000755419

читай эту книгу!

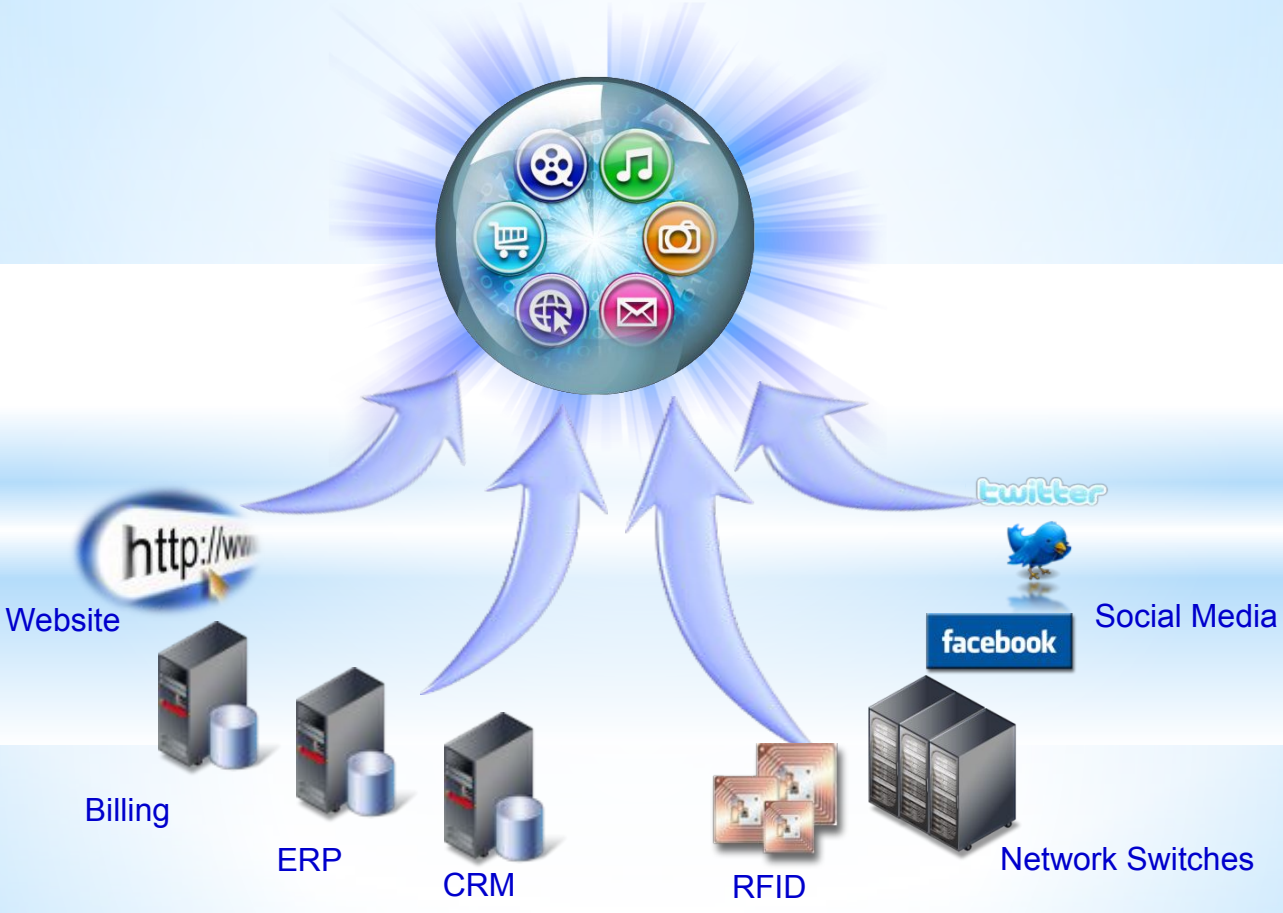
- Используется не отобранная малая часть данных (выборка), а ВСЁ доступное множество.

- Ради этого приходится отказываться от деталей (пренебрегать точностью) и исследовать не причинную связь (почему происходит явление), а корреляционные связи, объясняющие, что именно происходит (полезно, опасно).

Большие данные – горячая тема, потому что технологии сделали возможным анализ ВСЕХ доступных данных

Эффективно с точки зрения затрат управлять и анализировать

все доступные данные, в их первозданном виде – структурированные, неструктурированные, потоковые

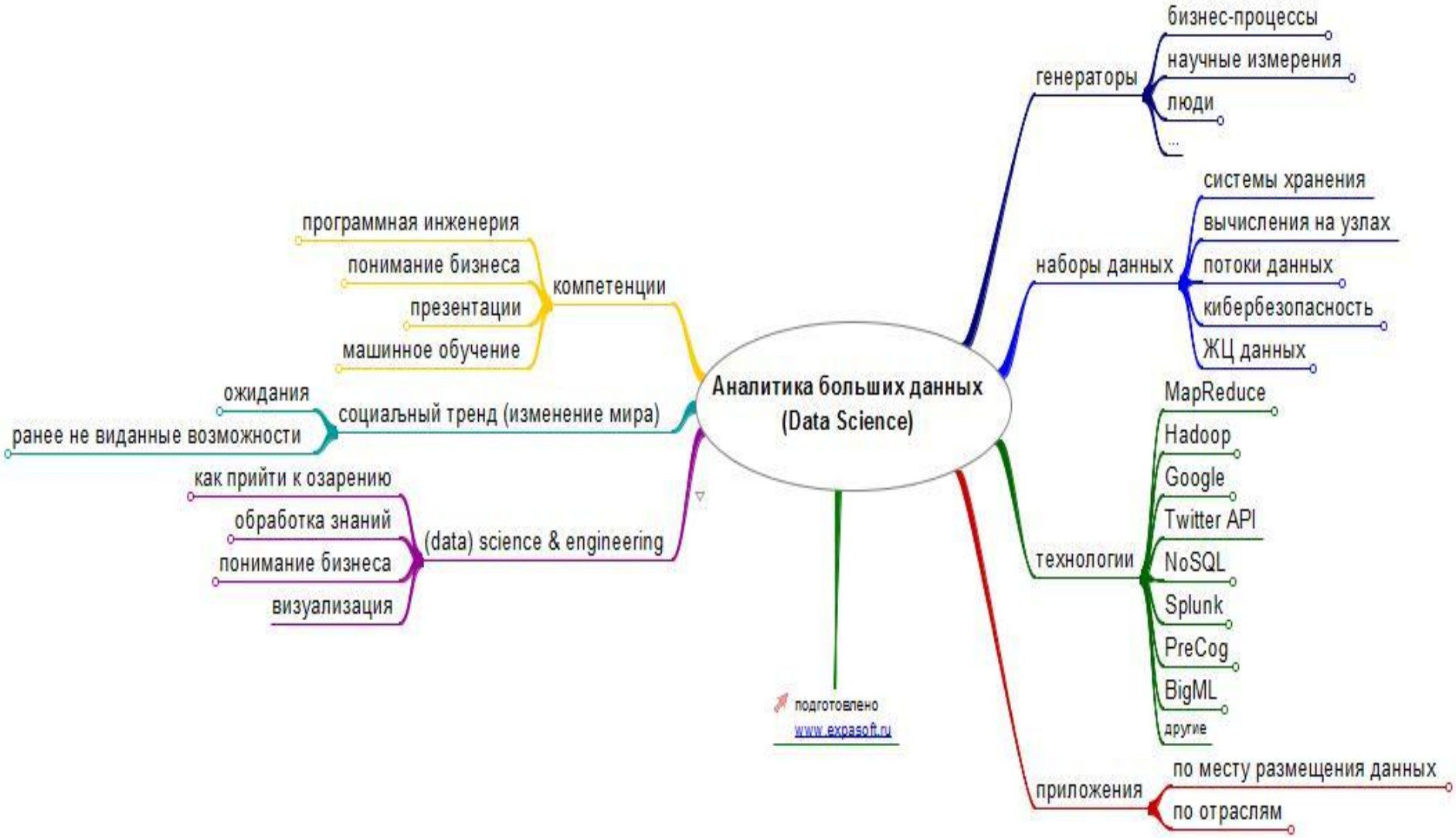


Обвал данных

- Каждый день в мире производится 2,5 квинтильона (10^{18}) байтов данных. 90% данных созданы за последние два года.
- Каждый час совершается 1 миллион сделок, пополняя базу данных на 2,5 петабайта (10^{15})- в 170 раз больше объема данных Библиотеки Конгресса США.
- Объем отправлений, доставляемых американской Почтовой службой за один год, равен 5 петабайтам, а Google обрабатывает такой же объем данных всего за один час.
- Суммарный объем всей существующей на земле информации составляет несколько больше одного

Прогноз роста данных до 2015 года

*Первый взгляд на большие данные



*Инженерный взгляд

- хранилища
  - системы хранения данных
  - облака
  - EMC
    - оборот
  - Oracle
  - IBM
  - Amazon
  - один админ на 10000 виртуальных
  - Cloudera
  - хранить уже дорого
    - зачем столько собираем?

- кибербезопасность
  - передача данных, политики, контроль
  - как обрабатывать?
    - контуры данных
    - контроль за копированием
    - права доступа
    - Утечки
    - шифрование/дешифрование

- жизненный цикл данных
  - создание (в т.ч. автоматическое)
  - обработка
  - анализ
    - систематизация
    - озарения
    - визуализация
  - отчёты
  - уничтожение
    - захоронения, как ядерные отходы
    - хранить дорого
      - а что хранить, что удалять?

- потоки данных
  - коммуникации
    - человек-человек
    - человек-компьютер-чел
    - человек-компьютер
    - компьютер-человек
    - компьютер-компьютер
  - пропускная способность
    - ограничивающий фактор

- готовые технологии обработки
  - Google FS
  - Hadoop
  - MapReduce

- вычисления на узлах, где данные собраны

ЕС-лизинг

Требования к функциям платформы Больших Данных

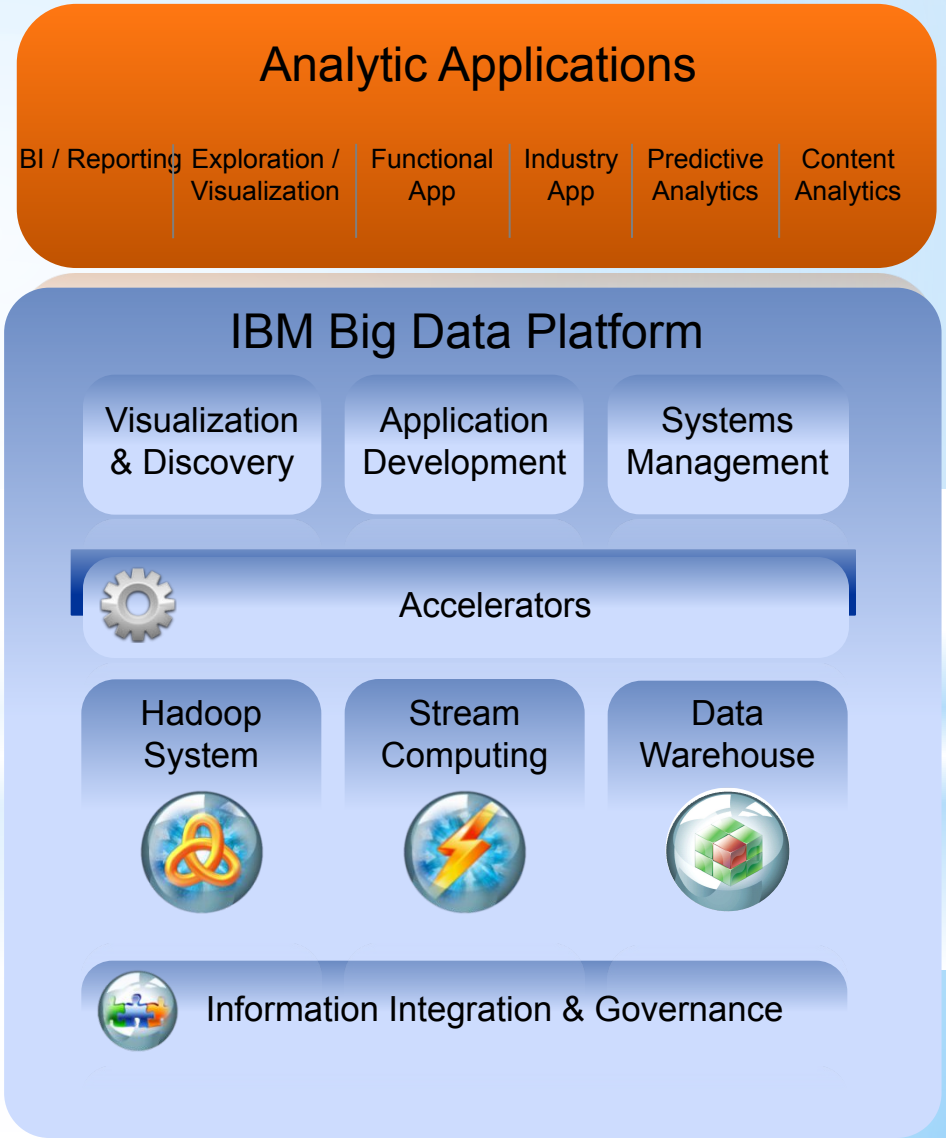
IBM

Поиск и навигация источников данных в киберпространстве		InfoSphere Data Explorer и т.д.
Подключение источников и анализ данных «в покое»		Hadoop File System и т.д.
Подключение источников и анализ данных «в движении»		InfoSphere Streams и т.д.
Традиционные функции работы со структурированными данными		Netezza и т.д.
Интеграция всех видов данных для комплексного анализа		IBM Information Server IBM Change Data Capture
Автоматизация принятия решений и построение гипотез и прогнозов		IBM Cognos IBM SPSS

Стратегия IBM Big Data : приблизить аналитику к данным

Новые аналитические приложения выдвигают требования к платформе big data:

- **Объединять и управлять всем разнообразием (Variety), скоростью (Velocity) и объемом (Volume), достоверностью (Veracity) и обоснованностью (Validity) данных**
- **Применять передовую аналитику к информации в ее исходной форме**
- **Визуализировать все доступные данные для специального анализа**
- **Среда проектирования для создания новых аналитических приложений**
- **Оптимизация рабочей нагрузки и планирование**
- **Безопасность и управление**



*Изменение парадигмы

Традиционный подход
Структурный и повторяемый анализ

Запомнил - обработал

Бизнес

Определяет что спросить

ИТ

Структурирует данные для ответа на вопрос

- Месячная отчетность
- Анализ прибыльности
- Анализ анкет

Ограничение: память

Большие данные
Итеративность и исследование

Обработал - запомнил

ИТ

Обеспечивает платформу для креативного анализа

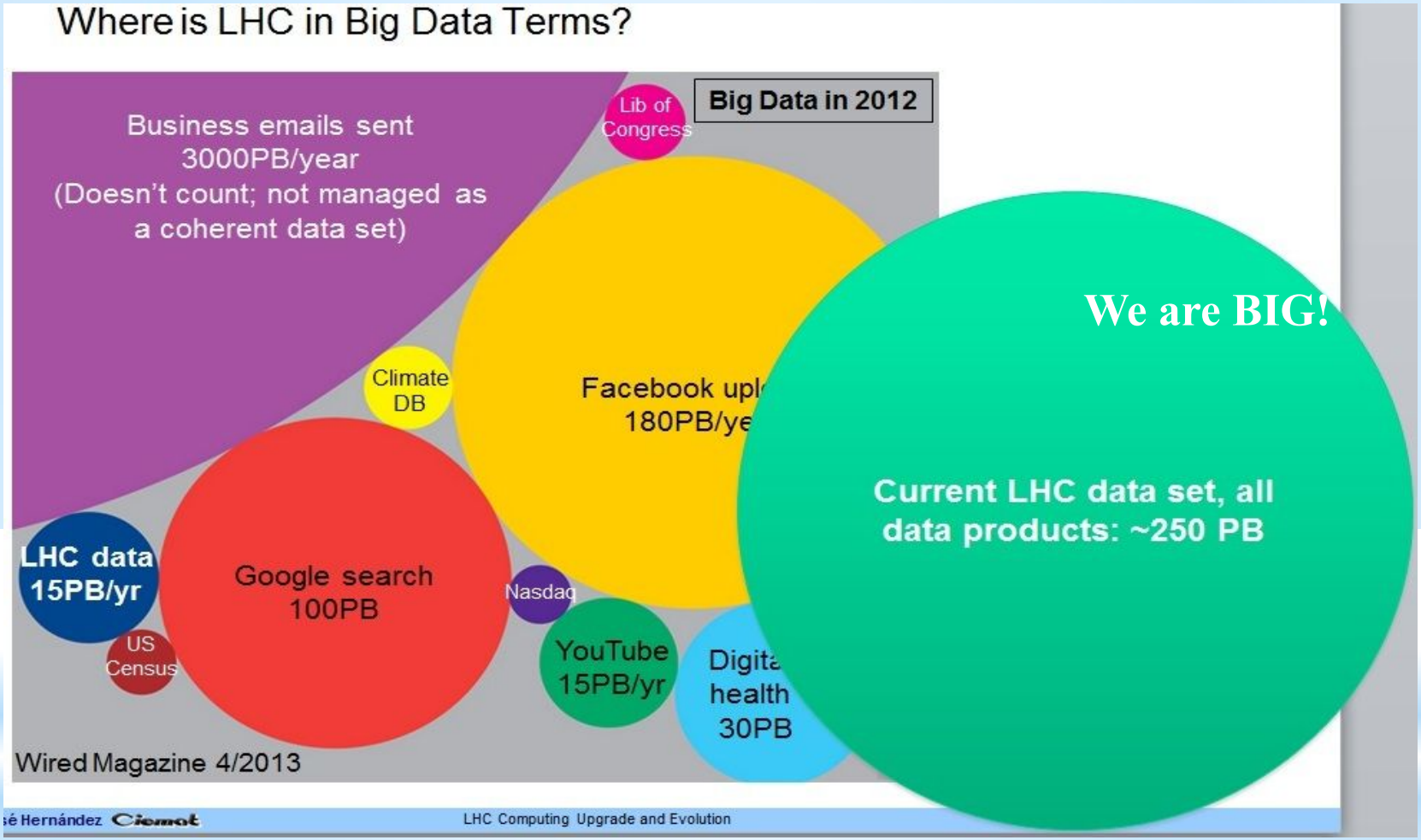
Бизнес

Исследует что можно спросить

- Отношение к бренду
- Стратегия продуктов
- Оптимизация ресурсов

Ограничение: производительность

*Вступление в эру Big Data



Сравнительная диаграмма обрабатываемых данных наглядно показывает, что исследования в ЦЕРНе идут в **условиях Больших Данных.**

После модернизации и запуска LHC в 2015 году поток данных возрастет в 2.5 раза, что потребует увеличения ресурсов и оптимизации их использования.

*CERN “Big Data”

Physics Data on CASTOR/EOS

- LHC experiments produce ~10GB/s 25PB/year

User Data on AFS & DFS

- Home directories for 30k users
- Physics analysis development
- Project spaces for applications

Service Data on AFS/NFS/Filers

- Databases, admin applications

Tape archival with CASTOR/TSM

- Physics data (RAW and derived/additional)
- Desktop/Server backups

Service	Size	Files
AFS	290 TB	2.3 B
CASTOR	89.0 PB	325 M
EOS	20.1 PB	160 M

*PanDa в эксперименте ATLAS

В эксперименте ATLAS на Большом адронном коллайдере разработана платформа для управления вычислительными ресурсами PanDA Workload Management System (WMS), которая обладает следующими возможностями:

- Проект PanDA начался в 2005 году группами BNL и UTA - **P**roduction **a**nd **D**ata **A**nalysis system.
- Автоматизированная и гибкая система управления заданиями, которая может оптимально сделать распределенные ресурсы доступными пользователю.
- С помощью PanDA, физики видят единый вычислительный ресурс, который предназначен для обработки данных эксперимента, даже если дата-центры разбросаны по всему миру
- PanDa изолирует физиков от аппаратного обеспечения, системного и промежуточного программного обеспечения и других технологических сложностей, связанных с конфигурированием сети и оборудования.
- Вычислительные задачи автоматически отслеживаются и выполняются. Могут выполняться групповые задачи физиков

В настоящее время PanDa контролирует:

- **сотни дата - центров в 50 странах мира**
- **сотни тысяч вычислительных узлов**
- **сотни миллионов заданий в год**
- **тысячи пользователей**

Интеграция технологий распределенных вычислений для управления большими данными

PanDa, как платформа, обеспечивающая прозрачность процесса хранения, обработки и управления данными для приложений с большими потоками данных и массивными вычислениями.

Развитие PanDA в направлении интеграции различных систем распределенных и параллельных вычислений (грид, cloud, кластеры, ЦОД, суперкомпьютеры) с целью создания универсальной платформы для крупных проектов управления большими данными в науке, государственном управлении, медицине, высокотехнологической промышленности, бизнесе.

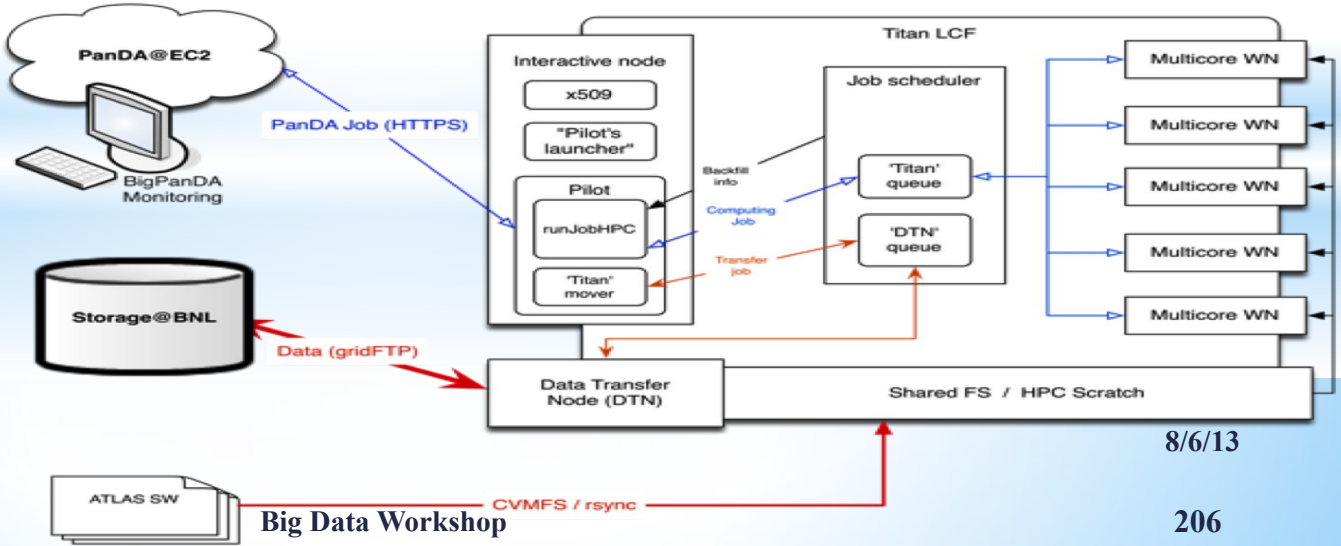
Суперкомпьютеры №15, 2013, стр.56

* Пример использования системы PanDA (суперкомпьютер Titan в Oak Ridge)

	Titan System (Cray XK7)		
Peak Performance	27.1 PF 18,688 compute nodes	24.5 PF GPU	2.6 PF CPU
System memory	710 TB total memory		
Interconnect	Gemini High Speed Interconnect	3D Torus	
Storage	Lustre Filesystem	32 PB	
Archive	High-Performance Storage System (HPSS)	29 PB	
I/O Nodes	512 Service and I/O nodes		

12 OLCF | 20

Конфигурация суперкомпьютера Titan и схема управления заданиями с помощью платформы PanDA



* Ресурсы, доступные с помощью платформы PanDA

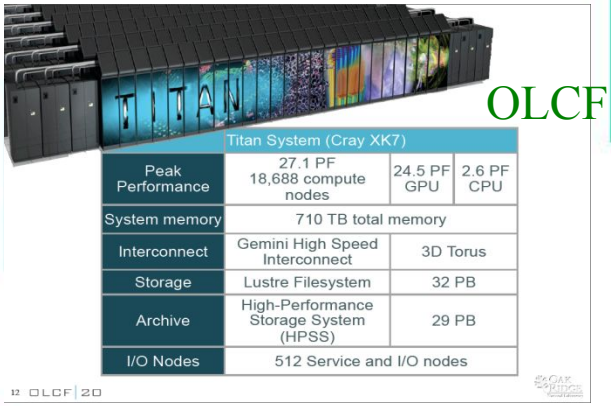
WLCG
Worldwide LHC Computing Grid

Many Others

NORDUGRID
Grid Solution for Wide Area Computing and Data Handling

Open Science Grid

nectar



OLCF

Titan System (Cray XK7)			
Peak Performance	27.1 PF 18,688 compute nodes	24.5 PF GPU	2.6 PF CPU
System memory	710 TB total memory		
Interconnect	Gemini High Speed Interconnect	3D Torus	
Storage	Lustre Filesystem	32 PB	
Archive	High-Performance Storage System (HPSS)	29 PB	
I/O Nodes	512 Service and I/O nodes		

12 OLCF | 20

FutureGrid

amazon web services™ EC2

Google Cloud Platform

Проекты с суперкомпьютерными центрами НИЦ КИ, ННГУ, Острава (Чехия)
Центры Больших данных в НИЦ КИ, ЛИТ ОИЯИ, РЭУ им. Плеханова

*Data Mining to deal with Big Data

Data Mining (DM) - “Это технология, которая предназначена для поиска в больших объемах данных неочевидных, объективных и полезных на практике закономерностей.” ***Григорий Пятецкий-Шапиро,*** *1989 г.*

Переводы DM на русский: **добыча** данных, **вскрытие** данных

Интеллектуальный анализ данных — это процесс обнаружения в сырых данных ранее неизвестных, нетривиальных, практически полезных и доступных интерпретации знаний, необходимых для принятия решений в различных сферах человеческой деятельности путем комбинации методов статистики и искусственного интеллекта с использованием технологии баз данных.

В современных условиях данных слишком много, они неоднородны, неполны, неструктурированны и содержат ошибки, а какой-либо рациональной теории для их описания, как правило, нет. Поэтому происходит сдвиг парадигмы их обработки от классической схемы моделирования на основе известной теории, а затем проверки модели сравнением с экспериментом традиционными средствами анализа данных к новой парадигме, когда модели, описывающие связи и зависимости создаются непосредственно из самих данных новыми средствами Data Mining.

Одно из основных положений Data Mining – поиск **неочевидных** закономерностей. Инструменты Data Mining могут находить такие закономерности самостоятельно и также самостоятельно строить гипотезы о взаимосвязях.

* Принципиальная схема системы сбора, хранения и анализа Больших данных