Проектирование архитектур программного обеспечения

лекция 7

Зозуля А.В.

КСПТ

2016 г.

ранее..

Типовые решения представления данных в Web

Монолитное web-приложение

- Стек программных компонентов:
  - HTTP-сервер + СУБД
  - Server & Client Side (MVC-фреймворк) — монолитный программный код
  - Чаще запущены на одном сервере
- Плюс: нет накладных расходов на интеркоммуникацию компонентов (сервисов)
- Минусы:
  - Высокая сложность разработки
  - Ошибка приводит к неработоспособности всего приложения
  - **Невозможность распараллелить разработку**
  - **Невозможно горизонтально масштабировать**

Сервис-ориентированная архитектура (SOA)

- ПО сайта разделяется на сервисы со строго определенной функциональностью: службы авторизации, сообщений, ленты новостей, поиска и пр.
- Сервисы имеют API обмениваются данными по определенному протоколу (REST API / DB)
- Плюсы:
  - **Возможность распределенной разработки**
  - **Горизонтальная масштабируемость**
  - Удобство тестирования сервисов
- Минусы:
  - Более сложная архитектура
  - Накладные расходы на интеркоммуникацию

Нагруженное web-приложение

- Большое количество запросов в единицу времени
- Операции с большими объемами данных
- Разнородность данных
- Высокая связность данных
- Работает в условиях отказов оборудования

Twitter - пример нагруженного web-приложения

- 600 млн сообщений в день
- 555 млн пользователей
- 135 тыс регистраций пользователей в день
- 9000 твитов в секунду
- 2 млрд поисковых запросов в день
- 25% трафика - веб сайт, остальное идет через API
- 10 млрд запросов к API в день, около 100 тыс в секунду
- 50 Гб новых данных в минуту
- Тысячи серверов
- Никаких облаков и виртуализации

Требования к нагруженному web-приложению

- **Масштабируемость**
- Способность «держать» нагрузку
- Легкость в администрировании
- Устойчивость к отказу оборудования
- Нечувствительность к оборудованию
- Исходный код не должен меняться при росте нагрузки

Масштабируемость (Scalability)

- Способность системы справляться с увеличением рабочей нагрузки (увеличивать свою производительность) при добавлении ресурсов (обычно аппаратных)
- Способность ПО корректно работать на малых и на больших системах с производительностью, которая увеличивается пропорционально вычислительной мощности системы
- Принцип построения открытых систем, гарантирующий сохранение инвестиций в информацию и ПО при переходе на более мощную аппаратную платформу

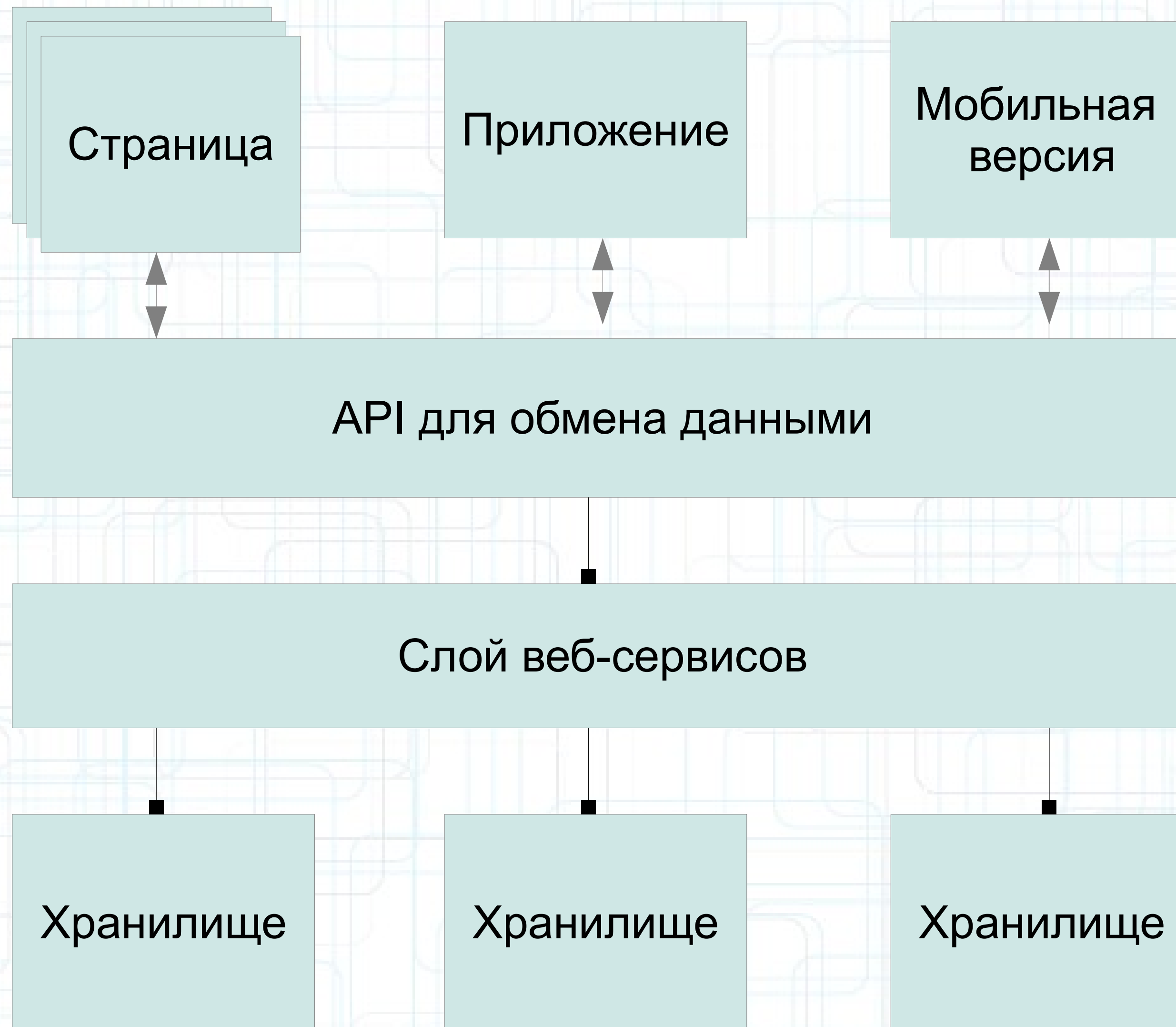
Масштабируемость

- Вертикальная
  - Наращивание производительности аппаратных узлов
  - Совершенствование кода и систем хранения
  - Дорого
- Горизонтальная
  - Наращивание числа аппаратных узлов
  - Миграция независимых программных компонентов
  - Миграция данных
  - Дешево, отвечает требованиям бизнеса

Подходы к разработке нагруженных приложений

- **Промышленный** - средства масштабирования разрабатываются отдельно от бизнес-логики: Facebook, Яндекс, Google. Разработчики сервисов не озабочены масштабированием, работают со слоем big data, как с «черным ящиком». Пример: сервис Google+ был создан за два месяца.
- **Ремесленный** - средства масштабирования и бизнес-логика разрабатываются одновременно: Вконтакте. Разработчик каждого сервиса озабочен вопросами масштабирования.

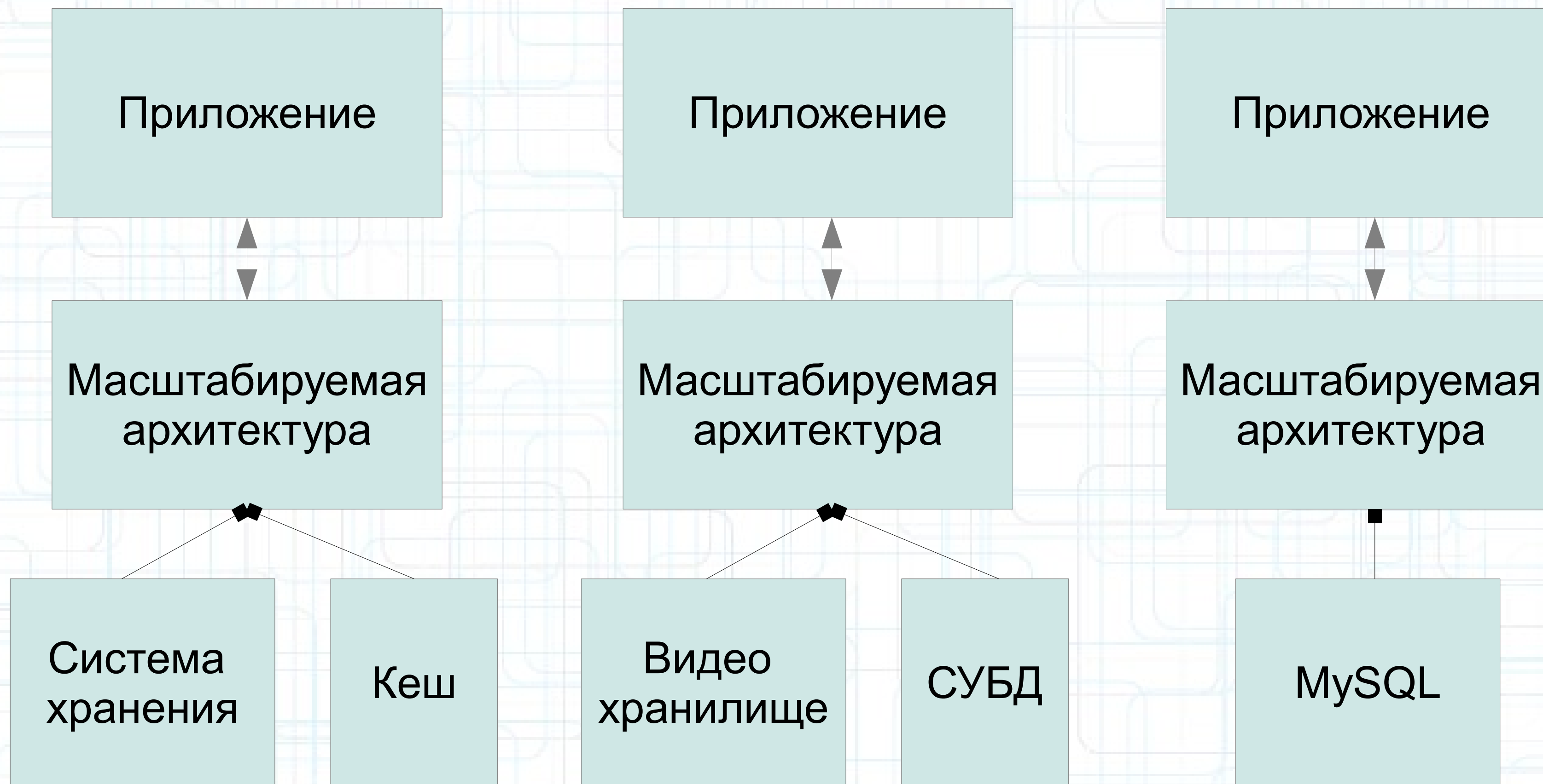
Промышленный подход



Промышленный подход

- Долгая разработка общих инструментов
- Быстрая разработка приложений
- Возможность использования для разработки приложений программистов средней и низкой квалификации — высокая масштабирумость разработки
- Повышенные требования к аппаратному обеспечению

Ремесленный подход



Ремесленный подход

- Быстрая разработка любых новых решений
- Высокие требования к квалификации разработчиков — низкая масштабирумость разработки
- Максимально эффективное использование технологий и аппаратного обеспечения

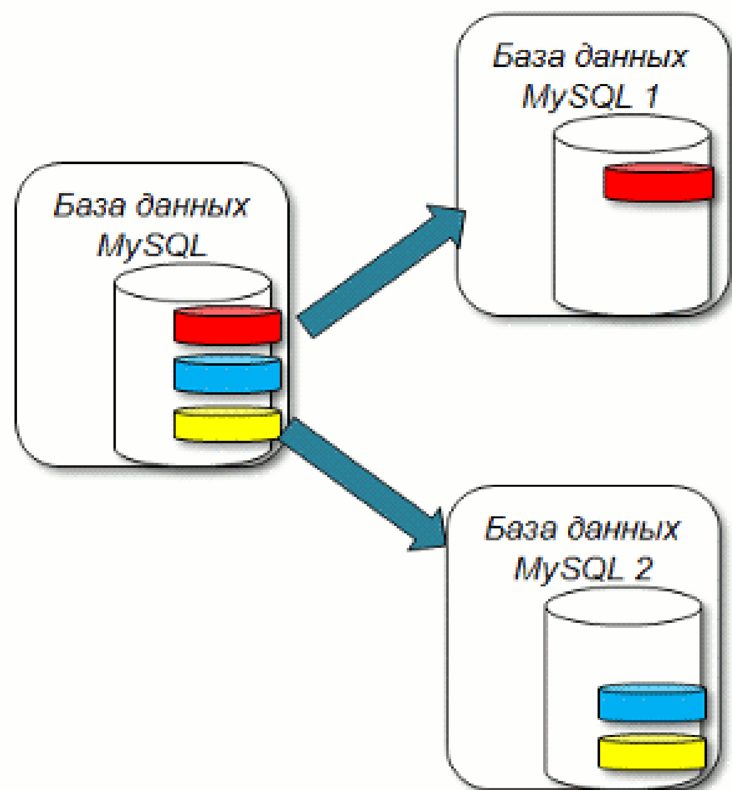
Компоненты нагруженных web-приложений

- HTTP-сервер-балансировщик
- Сервер приложений
- Сервер статических данных
- Система обработки очереди запросов
- Распределенный кеш
- Распределенная поисковая система
- Специализированные noSQL системы хранения
- Реляционная СУБД с шардингом

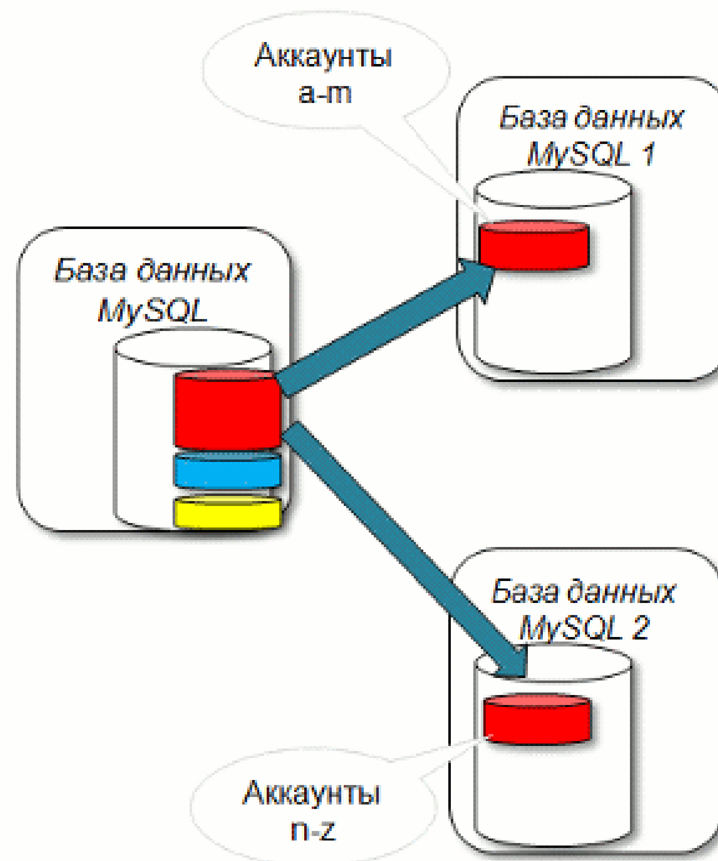
Шардинг

- «Дерзкая» альтернатива репликации
- Задача: устранить узкое место — доступ к одной БД
- Логическое разделение данных исходя из требование производительности
  - Вертикальный — выделение шардов по типам данных
  - Горизонтальный — выделение шардов по порциям и диапазонам данных

Шардинг



Вертикальный шардинг



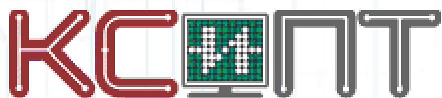
Горизонтальный шардинг

(c) dev.1c-bitrix.ru

Горизонтальный шардинг

```
class ShardingStrategy {
    protected static $instance = null;
    protected $server1;
    protected $server2;

    protected function __construct() {
        $this->server1 = mysql_connect('server1', 'user1', 'pass1');
        $this->server2 = mysql_connect('server2', 'user2', 'pass2');
    }

    public static function getInstance() {
        if (static::$instance == null) {
            static::$instance = new self();
        }

        return static::$instance;
    }

    public function getConnection(Order $order) {
        $server = $this->server1;
        if ($order->user_id % 2 == 0) $server = $this->server2;
        return $server;
    }
}
```

(c) phphighload.com



Шардинг

- Преимущества:
  - Легкость управления маленькими БД
  - Высокая скорость доступа
  - Нет необходимости в дорогостоящих СУБД
- Проблемы:
  - Определение «местоположения» шарда
  - Сложные JOIN-запросы
  - Определение нового идентификатора

Распределенный поиск

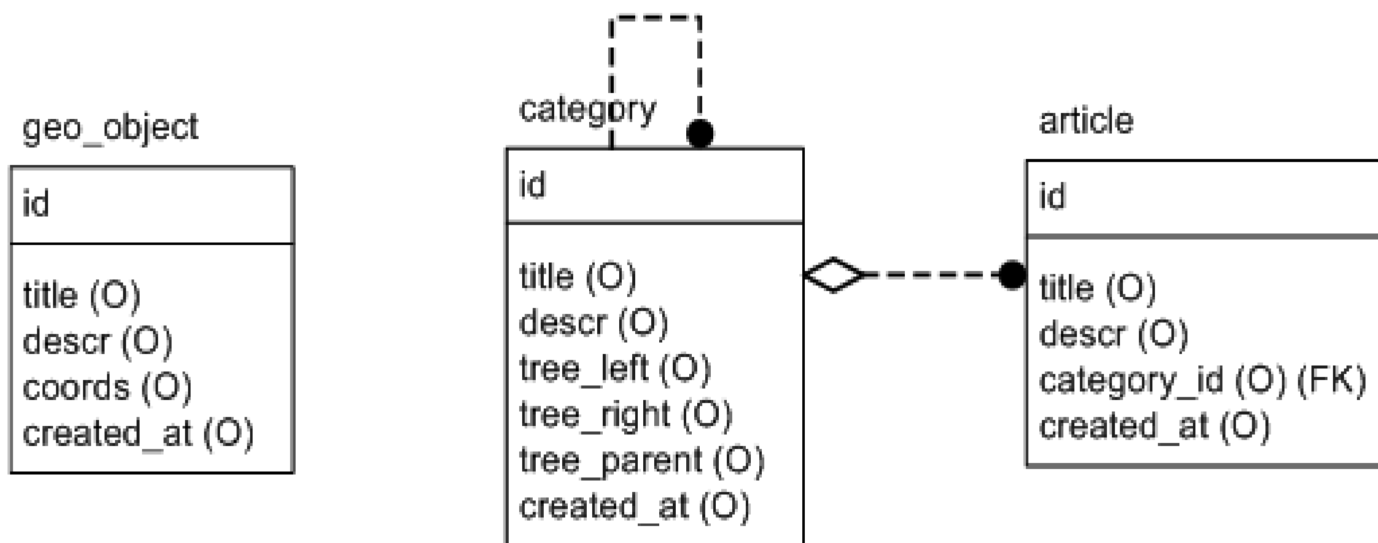
- LIKE-поиск имеет ограниченную функциональность
- Разнородные хранилища данных
- Распределенные хранилища данных
- Требуется полнотекстовый поиск => специализированные системы поиска
- Требуется регистрация каждого нового шарда
- Необходимо периодически перестраивать индекс
- Примеры поисковых систем:

Распределенный поиск Sphinx

- Высокая скорость индексации (до 10-15 МБ/сек на ядро);
- Высокая скорость поиска (до 150—250 запросов в секунду на каждое ядро с 1 000 000 документов);
- Высокая масштабируемость (крупнейший известный кластер индексирует до 3 000 000 000 документов и поддерживает более 50 миллионов запросов в день);
- Распределенная возможность поиска;
- Поддержка нескольких полей полнотекстового поиска
- Поддержка стоп-слов
- Поддержка однобайтовых кодировок и UTF-8
- Поддержка морфологического поиска
- Поддержка MySQL, PostgreSQL, ODBC совместимых СУБД

Распределенный поиск Sphinx

Задача: одновременный текстовый поиск по 3-ем типам объектов: географическим объектам, категориям информационной зоны и материалам информационной зоны — с возможностью фильтрации по дате публикации объектов и категориям, к которым они относятся.



(c) habrahabr.ru

Распределенный поиск Sphinx

```
#articles
source article
{   # параметры соединения с источником данных
    type = mysql   sql_host = localhost   sql_user = root   sql_pass = root   sql_db = ili_lv
    sql_query_range      = SELECT MIN(id), MAX(id) FROM article
    sql_range_step       = 500   #  делать выборку порциями по 500 записей
    sql_query_pre        = SET NAMES utf8
    # маска запроса, отправляемого Sphinx при индексации данных
    sql_query            =  SELECT id * 10 + 1 as id, category_id, 1 as row_type,\
         UNIX_TIMESTAMP(created_at) as created_at, title, descr \
         FROM article WHERE id >= $start AND id <= $end
   # описание атрибутов, которые можно использовать в качестве фильтров
    sql_attr_uint          = category_id
    sql_attr_uint          = row_type
    sql_attr_timestamp = created_at
    # маска запроса, извлекающего информацию по найденным id
    sql_query_info       = SELECT title, descr FROM article WHERE id = ($id - 1) / 10
}
```

```
 index site_search
{   # хранилища данных -  индекс формируется из нескольких source
    source = category      source = geo_object        source = article

    path              = /var/data/sphinx/site_search
    docinfo             = extern
    morphology           = stem_en, stem_ru    # морфологии
    html_strip          = 0                     # вырезание html-тегов
    charset_type       = utf-8                  # кодировка индекса
    min_word_len       = 2                      # минимальная длина слова
}
```

Распределенный поиск Sphinx

```
$sphinx = new sfSphinxClient($options);

// числовые фильтры
if ($request->getParameter('category_id')) {
   $sphinx->setFilter('category_id', array($request->getParameter('category_id')));
}
if ($request->getParameter('row_type')) {
   $sphinx->setFilter('row_type', array($request->getParameter('row_type')));
}
$dateRange = $request->getParameter('date');

// временные фильтры
if ($dateRange['from'] || $dateRange['to']) {
   $sphinx->setFilterRange('created_at',
                          !empty($dateRange['from']) ? strtotime($dateRange['from']) : '',
                          !empty($dateRange['to'])  ? strtotime($dateRange['to'])  : '');
}

$this->results = $sphinx->Query($request->getParameter('s'), 'site_search');
if ($this->results === false) {
   $this->message = 'Запрос не выполнен: ' . $sphinx->GetLastError();
} else {
   // извлечение информации по id индекса
   $this->items = $this->retrieveResultRows($this->results);
}
```

КСОПТ

Кеширование

- На одну операцию записи приходится 10 операций чтения
- Запрашиваемые данные могут хранится во множестве мест
- Требуется дублирование часто используемой информации => noSQL-хранилище ключ/значение
- Хранилище может быть как персистентным, так и временным
- Хранилище может быть как распределенным, так и локальным
- Пример хранилищ ключ/значение: Memcached, Membase, Redis, Cassandra, HBase

Кеширование Memcached

```
function get_foo(int userid) {
   /* вначале проверить кэш */
   data = memcached_fetch("userrow:" + userid);

   if (!data) {
      /*не найдено: запросить БД */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);

      /* сохранить в кэше для будущих запросов */
      memcached_add("userrow:" + userid, data);
   }

   return data;
}
```

```
function update_foo(int userid, string dbUpdateString) {
   /* вначале обновить БД */
   result = db_execute(dbUpdateString);
   if (result) {
      /*обновление БД состоялось: подготовить данные для занесения в кэш */
      data = db_select("SELECT * FROM users WHERE userid = ?", userid);

      /* занести обновленные данные в кэш */
      memcached_set("userrow:" + userid, data);
   }
}
```

Очереди запросов

- Невозможно выполнять все запросы синхронно (загрузка и обработка больших объемов данных)
- Запросы прежде выполнения должны сохраняться в очереди
- Используются также персистентные хранилища вида ключ/значение
- Примеры:

Kestrel

Реализация очереди на Redis

```
require "predis/autoload.php";
PredisAutoloader::register();

try {
    $redis = new PredisClient(array(
        "scheme" => "tcp",
        "host" => "127.0.0.1",
        "port" => 6379));
    echo "Successfully connected to Redis";

    $list = "PHP Frameworks List";
    $redis->rpush($list, "Symfony 2");              // appends element(s) to a list
    $redis->rpush($list, "Symfony 1.4");
    $redis->lpush($list, "Zend Framework");     // prepends element(s) to a list

    echo "Number of frameworks in list: " . $redis->llen($list) . "<br>";

    $arList = $redis->lrange($list, 0, -1);          // gets elements from a list
    echo "<pre>"; print_r($arList); echo "</pre>";

    echo $redis->rpop($list) . "<br>";              // the last entry in the list
    echo $redis->lpop($list) . "<br>";              // the first entry in the list
} catch (Exception $e) {
    echo "Couldn't connected to Redis: " . echo $e->getMessage();
}
```

Специализированные системы хранения

- Хранение и обработка древовидных данных (например, социальный граф)

  Пример: **FlockDB** - оптимизирован для работы с очень большими списками смежных вершин графов (10-ки миллиардов рёбер графов и поддерживает 10-ки тыс. операций записи и 100-ни тыс. операций чтения в сек.)

- Хранение больших объемов данных

  Пример: **HDFS** (Hadoop Distributed File System) - предназначена для хранения больших файлов, поблочно распределённых между узлами вычислительного кластера. Каждый блок может быть размещён на нескольких узлах. Размер блока и коэффициент репликации (количество узлов, на которых должен быть размещён каждый блок) определяются в настройках на уровне файла.

Статические данные

- Статические данные: медиа-контент, стили, javascript-файлы
- Занимают значительное количество дискового пространства
- Доступ к ним неоправданно нагружает сервер приложений
- => Отдельный сервер с легковесным web-сервером
- Пример: nginx, lighttpd

Балансировщик нагрузки

- Существует дорогостоящее специализированное сетевое оборудование
- Альтернатива: использование программного обеспечения
- ПО проксирует запросы между несколькими серверами
- При добавлении сервера нужно добавить несколько строк в конфигурационный файл
- Пример: nginx + mod_proxy, Apache + mod_proxy

Пример балансировщика nginx

```
upstream nextserver {        # список серверов для отдачи статических файлов
            server 192.168.0.2;
}
upstream backend {           # список серверов для отдачи динамического контента
            server 127.0.0.1;
}

server {
      listen *:80;
      server_name 173.194.32.2;

      location / {
            root       /var/www/default;      # корневой каталог сервера
            access_log   off;                 # отключаем access_log
            # проверяем существование файла, потом существование директории
            # если файл есть, то nginx отдает его клиенту
            # если нет - клиент перенаправляется на location @nextserver
            try_files $uri $uri/ @nextserver;
      }

      location @nextserver {
            proxy_pass          http://nextserver;  # перенаправляем запрос клиента
            proxy_connect_timeout      70;
            proxy_send_timeout         90;
      }

      location ~* \.(php5|php|phtml)$ {
            proxy_pass  http://backend;    # переадресовываем клиента обработчику
      }
}
```

Аппаратное обеспечение

- Унификация типового аппаратного узла
- Акцент на качество архитектуры, а не одного отдельного компонента
- Централизованные системы управления оборудованием. Пример: Loony
- Системы развертывания кода и ПО. Пример: Murder — обновление кода на 10 тысячах серверов за 1 минуту (протокол BitTorrent)
- Распределенная систем сбора и анализа файлов протокола. Пример: Scribe

Нагрузочное тестирование

- **Нагрузочный тест (Load-testing)** – определяется работоспособность системы при некоторой строго заданной заранее (планируемой, рабочей) нагрузке
- **Тест устойчивости (Stress Test)** – определяются минимально необходимые величины системных ресурсов для работы приложения, оцениваются предельные возможности системы, определяется способность системы к сохранению целостности данных при возникновении внештатных аварийных ситуаций.
- **Тест производительности (Performance Test)** – комплексная проверка, включающая предыдущие два теста, предназначена для оценки всех показателей системы.

Результаты теста: max число пользователей, которые могут одновременно получить доступ к веб-узлу, число запросов, обрабатываемых приложением, время ответа сервера.

далее..

Интеграция информационных систем

Промежуточное ПО Gizzard