Резюме по приложению Octet

Базовые технологии

Стандарт C++17

Обоснование: оптимальный баланс между современными возможностями и стабильностью:

- Встроенная библиотека `std::filesystem` для кроссплатформенной работы с файлами
- `std::string_view` для эффективной работы со строками без копирования
- Поддерживается всеми современными компиляторами на целевых платформах (Linux/macOS)

CMake 3.15+

Обоснование: стабильная версия с полной поддержкой C++17.

Архитектура приложения

Выбранный метод хранения: гибридное хранилище

- **В памяти приложения:** `std::unordered_map<std::string, std::string>` (GUID -> UTF-8 строка)
- **На диске:** периодические снимки состояния `std::unordered_map` + журнал операций (Write Ahead Logging)

Компоненты системы

1. **Менеджер хранилища** - ядро системы, управляющее данными в памяти и на диске
2. **Менеджер журнала** - журналирование и восстановление
3. **Генератор GUID** - создает уникальные идентификаторы для строк
4. **Процессор команд** - обрабатывает CLI-команды
5. **Утилиты файловой системы** - работа с файлами

Примерная структура файлов проекта

```
include/
├── storage/
│   ├── storage_manager.hpp
│   ├── journal_manager.hpp
│   └── guid_generator.hpp
├── commands/
│   └── command_processor.hpp
└── utils/
    ├── file_utils.hpp
    └── string_utils.hpp

src/
├── storage/
│   ├── storage_manager.cpp
│   ├── journal_manager.cpp
│   └── guid_generator.cpp
├── commands/
│   └── command_processor.cpp
└── utils/
    ├── file_utils.cpp
    └── string_utils.cpp

tests/
├── storage_tests.cpp
├── journal_tests.cpp
└── guid_tests.cpp

app/
└── main.cpp
```

Основные интерфейсы

⚠️ Пока что сильно приблизительный вид, накидал базу, к реализации еще не приступал, поэтому может сильно поменяться

1. Менеджер хранилища (StorageManager)

```
/**
 * @class StorageManager
```

```
 * @brief Управляет хранением UTF-8 строк и их идентификаторов.
 *
 * Реализует гибридное хранилище с данными в памяти и постоянное хранение на
диске.
 * Обеспечивает базовые операции вставки, получения, обновления и удаления.
 */
class StorageManager {
public:
    /**
     * @brief Конструктор с указанием путей к файлам хранилища.
     * @param dataDir Директория для хранения файлов.
     */
    explicit StorageManager(const std::filesystem::path& dataDir);

    /**
     * @brief Добавляет UTF-8 строку в хранилище.
     * @param data Строка данных для сохранения.
     * @return Уникальный идентификатор (GUID) для добавленной строки.
     */
    std::string insert(const std::string& data);

    /**
     * @brief Извлекает строку по её идентификатору.
     * @param guid Уникальный идентификатор строки.
     * @return Сохранённая строка данных или пустая строка, если не найдена.
     */
    std::string get(const std::string& guid);

    /**
     * @brief Обновляет существующую строку новыми данными.
     * @param guid Уникальный идентификатор строки для обновления.
     * @param data Новые данные для сохранения.
     * @return true если обновление выполнено успешно, false если строка не
найдена.
     */
    bool update(const std::string& guid, const std::string& data);

    /**
     * @brief Удаляет строку из хранилища.
     * @param guid Уникальный идентификатор строки для удаления.
     * @return true если удаление выполнено успешно, false если строка не
найдена.
     */
    bool remove(const std::string& guid);

    /**
```

```
     * @brief Создаёт снимок текущего состояния хранилища.
     * @return true если снимок создан успешно.
     */
    bool createSnapshot();

    /**
     * @brief Загружает данные из файлов в память.
     * @return true если загрузка выполнена успешно.
     */
    bool loadFromDisk();

private:
    // Хранилище в памяти
    std::unordered_map<std::string, std::string> dataStore;

    // Пути к файлам
    std::filesystem::path snapshotPath;

    // Менеджер журнала
    JournalManager journalManager;

    // Генератор GUID
    GuidGenerator guidGenerator;
};
```

2. Менеджер журнала (JournalManager)

```
/**
 * @class JournalManager
 * @brief Управляет журналом операций и обеспечивает восстановление данных.
 *
 * Реализует механизм журналирования (WAL) для обеспечения устойчивости
 * к сбоям и возможности восстановления данных после неожиданного
завершения.
 */
class JournalManager {
public:
    /**
     * @brief Типы операций для журнала.
     */
    enum class OperationType {
        INSERT,
        UPDATE,
        REMOVE,
        CHECKPOINT
```

```
    };

    /**
     * @brief Конструктор с указанием пути к файлу журнала.
     * @param journalPath Путь к файлу журнала операций.
     */
    explicit JournalManager(const std::filesystem::path& journalPath);

    /**
     * @brief Записывает операцию в журнал.
     * @param opType Тип операции.
     * @param guid Идентификатор строки.
     * @param data Данные операции (для INSERT и UPDATE).
     * @return true если запись выполнена успешно.
     */
    bool writeOperation(OperationType opType, const std::string& guid,
                        const std::string& data = "");

    /**
     * @brief Создаёт запись контрольной точки в журнале.
     * @param snapshotId Идентификатор снимка состояния.
     * @return true если запись выполнена успешно.
     */
    bool writeCheckpoint(const std::string& snapshotId);

    /**
     * @brief Воспроизводит операции из журнала для восстановления данных.
     * @param dataStore Ссылка на хранилище данных для восстановления.
     * @param lastCheckpoint Идентификатор последней контрольной точки
(опционально).
     * @return true если восстановление выполнено успешно.
     */
    bool replayJournal(std::unordered_map<std::string, std::string>&
dataStore,
                       const std::string& lastCheckpoint = "");

private:
    // Путь к файлу журнала
    std::filesystem::path journalFilePath;

    // Файловый поток для журнала
    std::fstream journalFile;

    /**
     * @brief Открывает файл журнала для записи.
     * @return true если открытие выполнено успешно.
```

```
     */
    bool openJournal();

    /**
     * @brief Обеспечивает синхронизацию данных с диском.
     * @return true если синхронизация выполнена успешно.
     */
    bool syncJournal();
};
```

3. Генератор GUID (GuidGenerator)

```
/**
 * @class GuidGenerator
 * @brief Генерирует уникальные идентификаторы для строк данных.
 *
 * Использует комбинацию временной метки, случайных чисел и счётчика
 * для создания глобально уникальных идентификаторов.
 */
class GuidGenerator {
public:
    /**
     * @brief Генерирует новый уникальный идентификатор.
     * @return Строка с новым GUID.
     */
    std::string generateGuid();

    /**
     * @brief Проверяет корректность формата GUID.
     * @param guid Проверяемый идентификатор.
     * @return true если формат корректен.
     */
    bool isValidGuid(const std::string& guid);

private:
    // Счётчик для обеспечения уникальности
    std::atomic<uint64_t> counter;

    // Генератор случайных чисел
    std::mt19937_64 rng;
};
```

4. Утилиты для работы с файлами (FileUtils)

```
/**
 * @namespace FileUtils
 * @brief Набор утилит для безопасной работы с файлами.
 */
namespace FileUtils {
    /**
     * @brief Атомарно записывает данные в файл.
     * @param filePath Путь к файлу.
     * @param data Данные для записи.
     * @return true если запись выполнена успешно.
     */
    bool atomicWrite(const std::filesystem::path& filePath, const
std::string& data);

    /**
     * @brief Безопасно читает данные из файла.
     * @param filePath Путь к файлу.
     * @param[out] data Буфер для прочитанных данных.
     * @return true если чтение выполнено успешно.
     */
    bool safeRead(const std::filesystem::path& filePath, std::string& data);

    /**
     * @brief Проверяет и восстанавливает повреждённый файл.
     * @param filePath Путь к файлу.
     * @param backupPath Путь к резервной копии (опционально).
     * @return true если восстановление не требуется или выполнено успешно.
     */
    bool checkAndRecoverFile(const std::filesystem::path& filePath,
                             const std::filesystem::path& backupPath = "");
}
```

Алгоритмы работы

1. Алгоритм выполнения команд работы с БД и журналом

1. **insert**:
   - Генерация нового GUID через `GuidGenerator::generateGuid()`
   - Запись операции в журнал через `JournalManager::writeOperation(INSERT, guid, data)`
   - Добавление записи в хеш-таблицу `dataStore[guid] = data`
   - Возврат сгенерированного GUID

2. **get**:
   - Поиск в хеш-таблице `dataStore.find(guid)`
   - Если найдено, возврат значения, иначе пустая строка
3. **update**:
   - Проверка существования GUID в хеш-таблице
   - Запись операции в журнал через `JournalManager::writeOperation(UPDATE, guid, data)`
   - Обновление значения в хеш-таблице `dataStore[guid] = data`
   - Возврат статуса операции
4. **remove**:
   - Проверка существования GUID в хеш-таблице
   - Запись операции в журнал через `JournalManager::writeOperation(REMOVE, guid)`
   - Удаление записи из хеш-таблицы `dataStore.erase(guid)`
   - Возврат статуса операции

2. Алгоритм загрузки данных при запуске

1. Проверка существования файлов (снимка и журнала)
2. Если найден снимок:
   - Загрузка снимка в хеш-таблицу `dataStore`
   - Определение ID последнего снимка
3. Восстановление операций из журнала через `JournalManager::replayJournal()`
   - Чтение журнала с последней контрольной точки
   - Последовательное применение операций к хеш-таблице
4. Создание нового снимка, если количество операций после последнего снимка превышает порог

3. Алгоритм надежной работы с файлами

1. **Атомарная запись файлов**:
   - Создание временного файла с новым содержимым
   - Установка прав доступа и владельца
   - Атомарное переименование для замены оригинального файла
   - Синхронизация директории (⚠️ лучше, конечно, использовать `fsync`, но можно и попроще)
2. **Создание снимка состояния**:
   - Запись заголовка снимка (метаданные, временная метка)

- Запись количества записей
- Последовательная запись пар GUID + данные
- Синхронизация файла (`fsync`)
- Запись контрольной точки в журнал

3. **Восстановление после сбоя**:
   - Проверка целостности файлов через контрольные суммы
   - Восстановление из резервных копий при необходимости
   - Обработка незавершенных операций журнала

4. Алгоритм генерации GUID

1. Получение временной метки с высоким разрешением
2. Генерация случайного компонента через криптографически стойкий генератор
3. Добавление атомарно увеличиваемого счетчика для уникальности
4. Комбинирование и кодирование компонентов в строку формата UUID

Расположение файлов хранилища

- **Linux**: `~/.local/share/octet/`
  - `data.snapshot` - файл снимка состояния
  - `operations.journal` - журнал операций
- **macOS**: `~/Library/Application Support/octet/`
  - `data.snapshot` - файл снимка состояния
  - `operations.journal` - журнал операций

Автоматическое тестирование

1. **Модульные тесты**:
   - `StorageManager` - проверка базовых операций
   - `JournalManager` - проверка записи и воспроизведения журнала
   - `GuidGenerator` - проверка уникальности и формата GUID
2. **Интеграционные тесты**:
   - Сценарии восстановления после сбоя
   - Стресс-тесты для большого объема данных
   - Тесты производительности

Заключение

Предложенная архитектура обеспечивает:

- **Максимальную производительность** за счет хранения данных в памяти приложения
- **Надежность** благодаря журналированию и атомарным операциям
- **Восстанавливаемость** после сбоев через механизм WAL
- **Оптимизированная работа с UTF-8 строками**, что позволяет абстрагироваться от их содержимого, и что делает её универсальной для различных типов данных