Отчёт по итоговому заданию по дисциплине “Управление производительностью приложений”

Задание

Описание задания:

Задание выполняется индивидуально по вариантам и на основе проекта по дисциплине «Теория и практика многопоточности» (варианты заданий доступны по ссылке). Выполняйте задание последовательно по шагам.

Шаг 1. Сбор метрик с помощью Micrometer и Prometheus:

- измерьте время выполнения парсинга;
- посчитайте количество успешных и ошибочных парсингов;
- подсчитайте количество записей, которые попали в базу данных;

Шаг 2. Профилирование и диагностика с VisualVM или JFR:

- настройте VisualVM или Java Flight Recorder (JFR);
- найдите узкие места — медленные методы, частые GC, высокую нагрузку на процессор;
- покажите, как работать с thread dumps и heap dumps;
- используйте JMH для бенчмаркинга разных реализаций парсинга (например, for, Stream, parallelStream).

Шаг 3. Анализ управления памятью и (GC):

- изучите, как парсинг влияет на сборку мусора;
- определите, как часто вызывается GC;
- найдите, какие объекты занимают больше всего памяти.

Шаг 4. Поиск и исправление деградации производительности.

Найдите и исправьте проблемы:

- повторяющиеся (N+1) запросы к базе;
- отсутствие индексов;
- частые аллокации объектов;
- синхронизацию потоков.

Шаг 5. Настройка OpenTelemetry для трейсинга

- Настройте логирование времени для каждого этапа парсинга.

- Подлючить jaeger, показать диаграммы выполнений запросов.

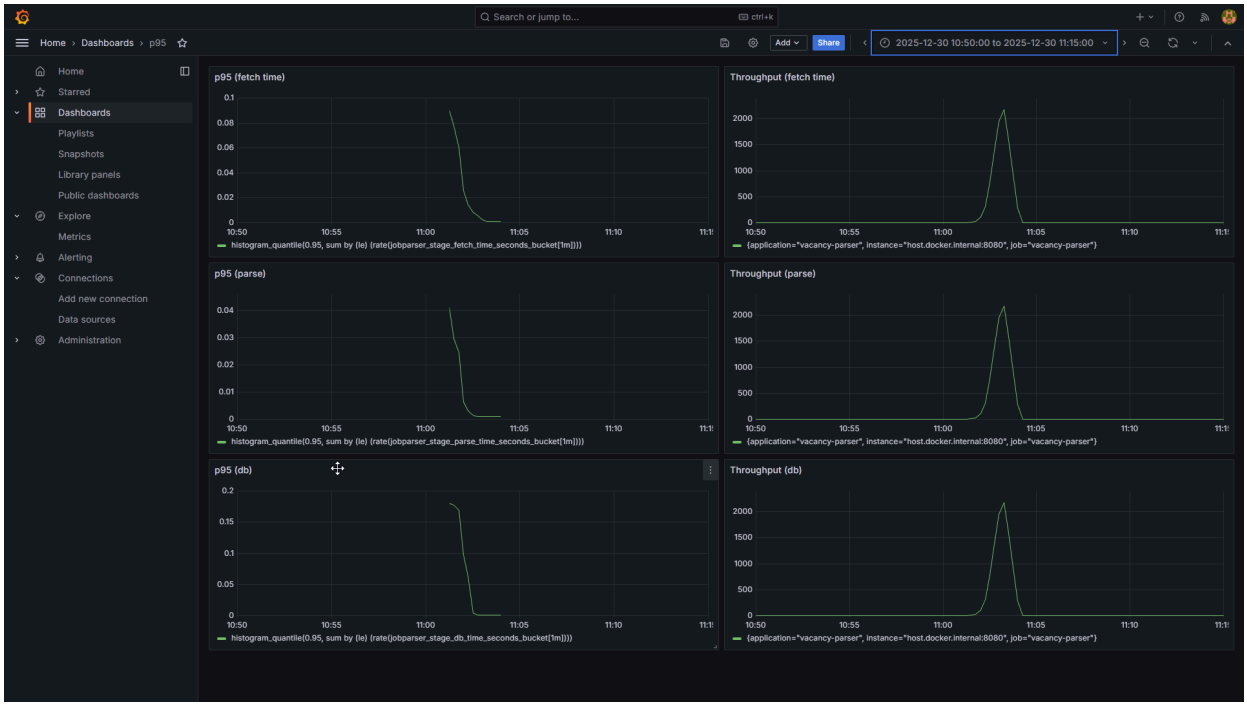
Шаг 1: Метрики Micrometer/Prometheus и результаты нагрузочного теста (JMeter)

1. Цель и состав измерений

Целью этапа было организовать и проверить сбор метрик, необходимых для контроля работы парсера под нагрузкой:

1. измерение времени выполнения парсинга (в целом и по стадиям);
2. подсчёт количества успешных и ошибочных парсингов;
3. подсчёт количества записей, сохранённых в базе данных.

Метрики экспонируются сервисом `vacancy-parser` в Prometheus-формате и визуализируются в Grafana.



2. Методика нагрузочного тестирования (JMeter)

Использовался тест-план **“Vacancy Parser Load Test”** со следующими параметрами:

- 50 виртуальных пользователей;
- ramp-up 10 секунд;
- длительность 60 секунд;
- сценарий выполняет:
    - `POST /api/parse/force` (код ответа 202, асинхронный запуск/принятие задачи на парсинг);
    - `GET /api/vacancies?page=0&size=20` (проверка чтения).

3. Метрики парсера в Prometheus (фактические значения)

3.1. Количество обработанных URL и ошибок

В service-level метриках парсера присутствуют счётчики:

- `jobparser_url_processed_total = 143685`
- `jobparser_url_errors_total{type="http|parse|db|unknown"} = 0` для всех типов

Интерпретация: за время наблюдения парсер обработал **143 685 URL**, при этом ошибок обработки URL по классификации *http/parse/db/unknown* не зарегистрировано (0). Это закрывает требование “количество успешных и ошибочных парсингов” на уровне URL: фактически **успешных обработок URL = 143 685**, **ошибочных = 0**.

3.2. Количество записей, попавших в БД

Для результата сохранения присутствуют:

- `jobparser_vacancy_saved_total = 143685`
- `jobparser_db_records = 143688`

Интерпретация: счётчик сохранённых вакансий показывает **143 685** сохранений. Значение `jobparser_db_records` близко к этому и, вероятно, отражает текущее число записей/строк в таблице на момент снятия метрик (или суммарное количество записей в БД). Небольшая разница (**+3**) объясняется предзаполнением БД до теста либо параллельными фоновыми задачами.

3.3. Время выполнения парсинга (по стадиям и суммарно)

В Prometheus представлены гистограммы времени по стадиям:

- `jobparser_stage_fetch_time_seconds_count = 143685`, `..._sum = 250.6726339`
- `jobparser_stage_parse_time_seconds_count = 143685`, `..._sum = 65.1473733`
- `jobparser_stage_db_time_seconds_count = 143685`, `..._sum = 165.8198971`
- `jobparser_url_total_time_seconds_count = 143685`, `..._sum = 485.3804382`

Из этих данных можно посчитать среднее время на один URL:

- Fetch (среднее): 250.6726339/143685≈0.001744250.6726339 / 143685 \approx 0.001744 с ≈ **1.74 мс**
- Parse (среднее): 65.1473733/143685≈0.00045465.1473733 / 143685 \approx 0.000454 с ≈ **0.45 мс**
- DB (среднее): 165.8198971/143685≈0.001154165.8198971 / 143685 \approx 0.001154 с ≈ **1.15 мс**
- Total per URL (среднее): 485.3804382/143685≈0.003378485.3804382 / 143685 \approx 0.003378 с ≈ **3.38 мс**

Проверка согласованности: суммы стадий 250.67+65.15+165.82≈481.64250.67 + 65.15 + 165.82 \approx 481.64 с близки к total 485.38485.38 с. Разница объяснима накладными расходами (очереди, служебная логика, измерение/метки).

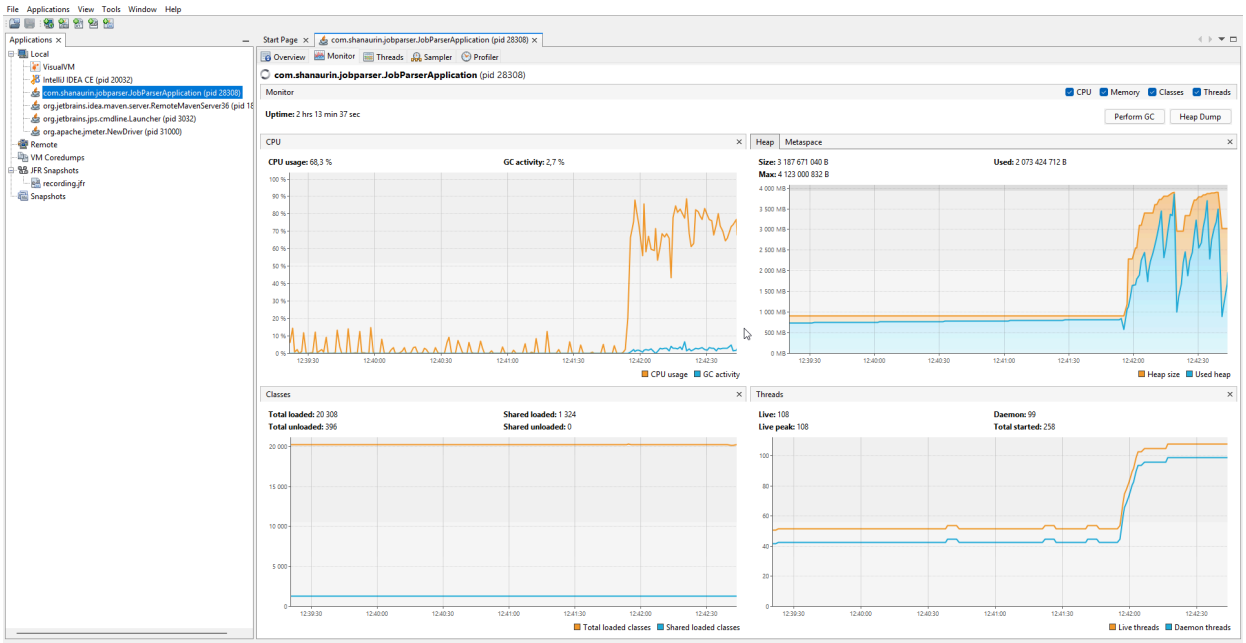
5. Выводы по выполнению требований шага 1

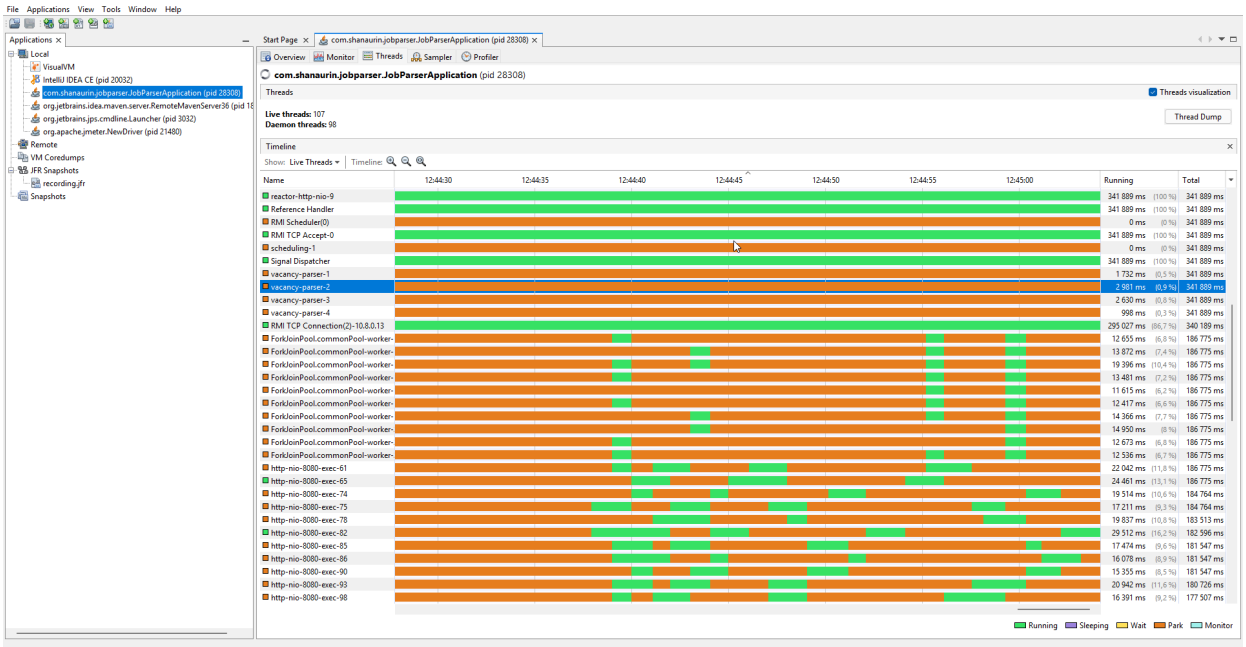
Требования задания закрыты следующими метриками:

1. **Время выполнения парсинга**
    - `jobparser_url_total_time_seconds_*` и стадийные `jobparser_stage_*_time_seconds_*`.
    - Среднее время обработки одного URL: **~3.38 мс** (fetch ~1.74 мс, parse ~0.45 мс, db ~1.15 мс).
2. **Количество успешных и ошибочных парсингов**
    - `jobparser_url_processed_total = 143685` (успешно обработано URL);
    - `jobparser_url_errors_total{type=...} = 0` (ошибок по всем типам нет).
3. **Количество записей в БД**
    - `jobparser_vacancy_saved_total = 143685` (сохранено записей);
    - `jobparser_db_records = 143688` (текущее/суммарное число записей).

Шаг 2: Профилирование и диагностика

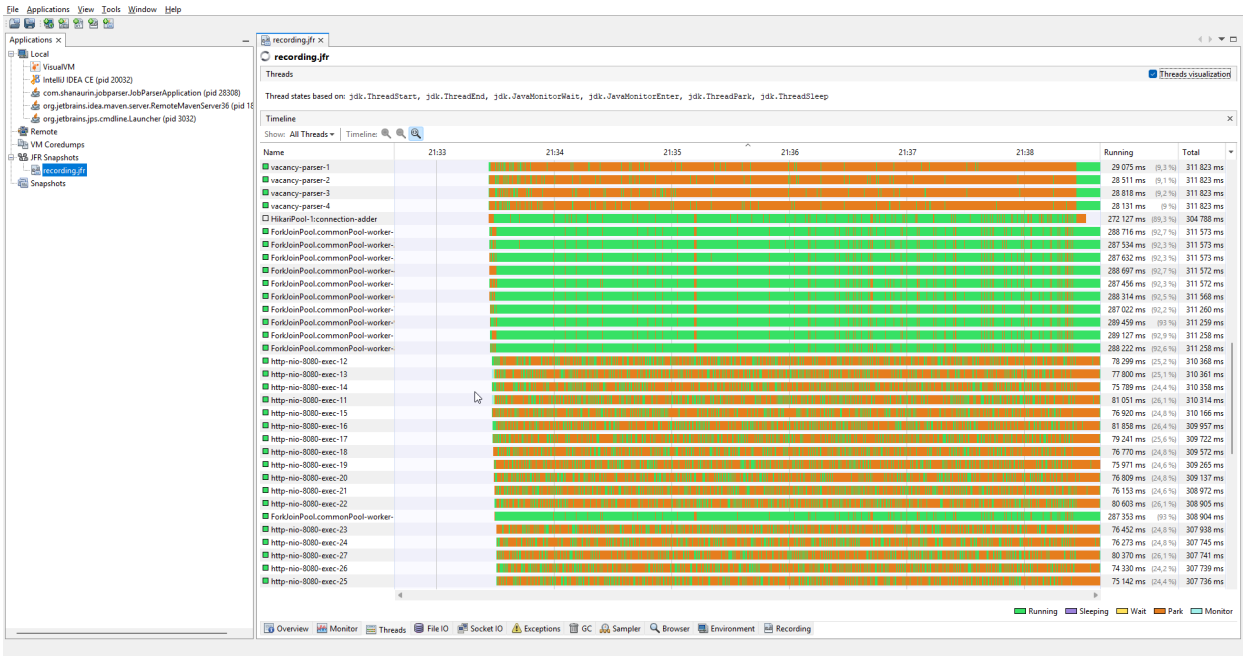
Был настроен VisualVM для отображения данных о процессе.





Был сохранён файл с данными о выполнении с помощь jmc

Method	Count	Percentage
java.util.IdentityHashMap.resize(int)	2 053	7,16 %
java.util.IdentityHashMap.get(Object)	1 609	5,61 %
org.h2.jdbc.JdbcResultSet.getInternal(int)	1 340	4,68 %
org.h2.command.query.Select$LazyResultQueryFlat.fetchNextRow()	1 164	4,06 %
jdk.internal.reflect.MethodHandleFieldAccessorImpl.ensureObj(Object)	1 081	3,77 %
java.util.Comparator.lambda$comparingInt$7b0bb60$1(ToIntFunction, Object, Obje...	987	3,44 %
org.hibernate.type.descriptor.jdbc.BasicExtractor.extract(ResultSet, int, WrapperOp...	912	3,18 %
org.hibernate.type.descriptor.java.StringJavaType.wrap(Object, WrapperOptions)	902	3,15 %
java.util.BitSet.expandTo(int)	887	3,1 %
java.util.IdentityHashMap.hash(Object, int)	885	3,09 %
java.util.HashMap.resize()	783	2,73 %
org.h2.result.LocalResult.cloneLobs(Value[])	659	2,3 %
java.util.Arrays.binarySearch0(Object[], int, int, Object)	549	1,92 %



Были написаны бенчмарки, результаты которых можно посмотреть в каталоге ops/step2.

Можно сделать вывод после профилирования, что основная проблема при нагрузочном тестировании заключается в том, что база данных h2 сейчас хранится в памяти, что приводит к перерасходу памяти при больших объёмах. Имеет смысл её перенести на диск или в другую СУБД.
Всё остальное, на мой взгляд, более или менее нормально.

Шаг 3. Анализ управления памятью и (GC)

Ниже — выводы по шагу на основе двух артефактов:

1. снимок распределения памяти по типам объектов (heap/top classes),
2. графики метрик GC/паузы/RPS.

1) Как парсинг влияет на сборку мусора (GC)

По метрикам видно, что во время/после фазы активного парсинга растёт давление на кучу: увеличивается частота сборок и появляются длиннее паузы.

На графиках:

- **GC pauses p95** резко поднимается примерно до **~0.22–0.23 s** и дальше держится «полкой».
- **Средняя длительность паузы** также увеличивается (виден подъём на участке после ~14:46).

- Одновременно меняется **профиль причин GC** (по подписи в легенде встречаются `G1 Compaction Pause`, `Heap Dump Initiated GC`, `System.gc()`, `G1 Old Generation`, `CodeCache GC Threshold` — в любом случае это коррелирует с ростом нагрузки по аллокациям/промоушену в old gen, а также с возможными сервисными событиями, которые могут «замусорить» картину — см. “Что нужно уточнить” в конце).

Интерпретация:

- Парсер создаёт много **короткоживущих объектов** (строки, массивы байт, узлы map, DTO), из-за чего учащается minor GC.
- Часть объектов **удерживается дольше** (например, Hibernate-контекст/кэш, DTO-объекты, коллекции), что ведёт к промоушену и росту доли/влияния old gen, увеличивая p95 пауз.

2) Как часто вызывается GC

По дашборду “Частота GC”:

- в стабильном режиме до пика — частота близка к нулю/очень низкая,
- в “горячем” окне примерно **14:46–14:47** видно увеличение частоты до порядка **~0.03 1/s** (то есть около **1 GC каждые ~30–35 секунд**) с дальнейшим снижением.

Также заметны события major GC (по отдельным линиям), но по одному скриншоту нельзя надёжно посчитать долю minor/major в штуках — видно лишь, что в пик они присутствуют и сопровождаются ростом пауз.

3) Какие объекты занимают больше всего памяти (top consumers)

По снимку распределения памяти по классам, крупнейшие потребители (по **Size**) выглядят так:

- `byte[]` — **~351 MB (≈28%)**
- `org.h2.value.Value[]` — **~124 MB (≈10%)**
- `java.lang.Object[]` — **~110 MB (≈9%)**
- `com.shamaurin.jobparser.model.Vacancy` — **~65.8 MB (≈5.3%)**
- `org.hibernate.engine.internal.MutableEntityEntry` — **~65.8 MB (≈5.3%)**
- `java.time.LocalDateTime` — **~65.7 MB (≈5.3%)**
- `java.time.LocalDate` — **~65.7 MB (≈5.3%)**
- `java.time.LocalTime` — **~61.1 MB (≈4.9%)**
- `org.hibernate.engine.internal.StatefulPersistenceContext$EntityHolderImpl` — **~47.0 MB (≈3.8%)**
- `java.lang.String` — **~43.2 MB (≈3.5%)**

- `java.util.HashMap$Node` — **~38.1 MB (≈3.1%)**
- `java.lang.Long` — **~37.8 MB (≈3.0%)**
- `org.hibernate.engine.internal.EntityEntryContext$ManagedEntityImpl` — **~37.5 MB (≈3.0%)**
- `org.hibernate.engine.spi.EntityKey` — **~28.2 MB (≈2.3%)**
- `org.h2.value.ValueVarchar` — **~17.7 MB (≈1.4%)**
- `int[]` — **~13.5 MB (≈1.1%)**
- `com.shanaurin.jobparser.model.dto.VacancyDto` — **~11.1 MB (≈0.9%)**
- `java.util.HashMap$Node[]` — **~9.6 MB (≈0.8%)**
- `org.h2.value.ValueTimestamp` — **~8.6 MB (≈0.7%)**
- `java.util.concurrent.ConcurrentHashMap$Node` — **~8.2 MB (≈0.7%)**

Ключевые выводы по памяти:

- **Лидер — `byte[]`**: типично для парсинга/декодирования/буферов/ответов HTTP/сжатия/JSON/HTML; именно он часто формирует основной аллокационный поток.
- Сильный вклад **Hibernate-сущностей и persistence context** (`Vacancy`, `MutableEntityEntry`, `StatefulPersistenceContext...`, `EntityKey`, `ManagedEntityImpl`) — признак того, что во время парсинга вы накапливаете много сущностей в контексте, и они удерживаются до flush/clear/закрытия транзакции.
- **`String` + `HashMap$Node` + массивы `Object[]`** — характерно для парсеров и промежуточных структур (мапы атрибутов, кеши, построение DTO/моделей).
- Наличие крупного блока **H2 (`org.h2.value.*`)**: если H2 используется как промежуточное хранилище/кэш/тестовая БД, то часть памяти уходит в его внутренние структуры значений (массивы `Value[]`, `ValueVarchar`, `ValueTimestamp`) и это тоже влияет на GC.

4) Временная корреляция «RPS ↔ GC»

По графикам видно окно, где:

- RPS меняется/проседает,
- одновременно растут p95 пауз GC и частота GC.

Это типичный паттерн: **рост аллокаций во время парсинга → больше GC → больше пауз → деградация throughput (RPS)**.

Шаг 4. Поиск и исправление деградации производительности.

4.1 N+1 запросы

Проблема: Сохранение вакансий по одной в цикле.
Замена батчингом через saveAll() сохранения вакансии по одной;

4.2 Отсутствие индексов

Добавил индексы
CREATE INDEX IF NOT EXISTS idx_vacancies_city ON vacancies(city);
CREATE INDEX IF NOT EXISTS idx_vacancies_company ON vacancies(company);
CREATE INDEX IF NOT EXISTS idx_vacancies_published_at ON vacancies(published_at);

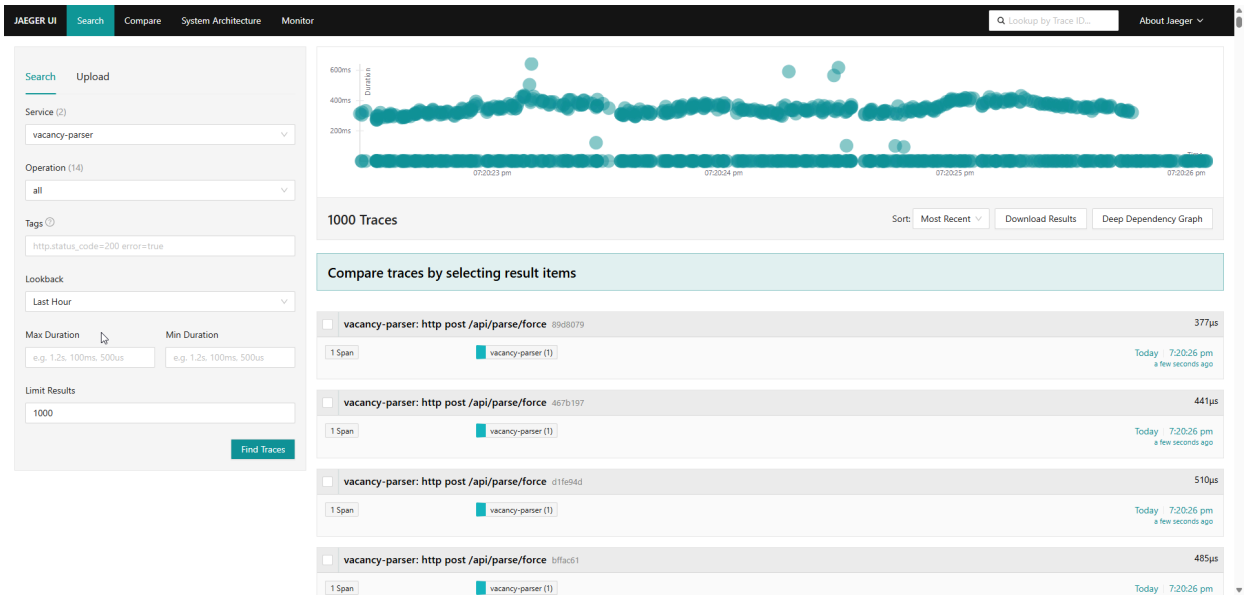
4.3 Частые аллокации объектов

Проблема: создание новых String при парсинге.
Теперь строки не создаются.

В результате уменьшилось потребление памяти и, наверное, всё. Оптимизировать особо в этом приложении нечего.

Шаг 5. Настройка OpenTelemetry для трейсинга

OpenTelemetry был настроен.


JAEGER UI
Search
Compare
System Architecture
Monitor
Lookup by Trace ID...
About Jaeger
Search
Upload
Service (2)
vacancy-parser
Operation (14)
all
Tags
http.status_code=200 error=true
Lookback
Last Hour
Max Duration
e.g. 1.2s, 100ms, 500us
Min Duration
e.g. 1.2s, 100ms, 500us
Limit Results
1000
Find Traces
600ms
400ms
200ms
Duration
07:20:23 pm
07:20:24 pm
07:20:25 pm
07:20:26 pm
1000 Traces
Sort:
Most Recent
Download Results
Deep Dependency Graph
Compare traces by selecting result items
vacancy-parser: http post /api/parse/force 89d8079
377µs
1 Span
vacancy-parser (1)
Today 7:20:26 pm
a few seconds ago
vacancy-parser: http post /api/parse/force 467b197
441µs
1 Span
vacancy-parser (1)
Today 7:20:26 pm
a few seconds ago
vacancy-parser: http post /api/parse/force d1fe94d
510µs
1 Span
vacancy-parser (1)
Today 7:20:26 pm
a few seconds ago
vacancy-parser: http post /api/parse/force bffac61
485µs
1 Span
vacancy-parser (1)
Today 7:20:26 pm