OZONtech

Тритон-сервер.

Ускорение моделей

Подготовил: Облаков Даниил
ML-инженер ОзонТех

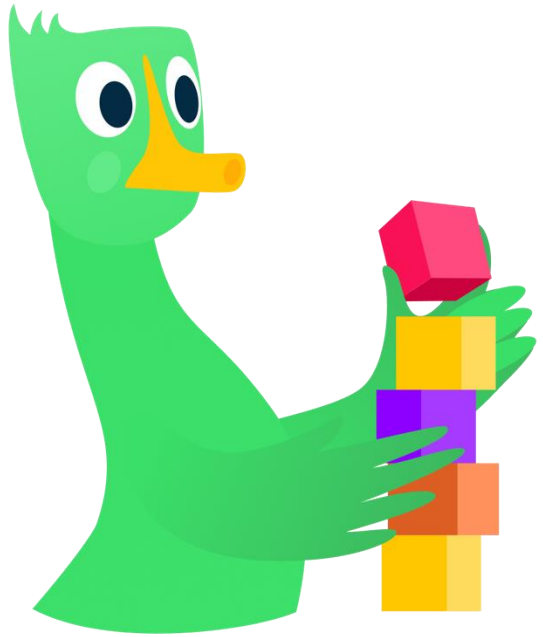
00

Откуда брать ускорение?

Точки оптимизации

- Модель
  - Компиляция
  - Оптимизации графа вычислений (Склейка слоев)
  - Квантизация
- Тритон
  - Батчевание
  - Instance-параллелизм
  - Кэширование
- Клиент
  - Асинхронные запросы
  - Потоковые запросы

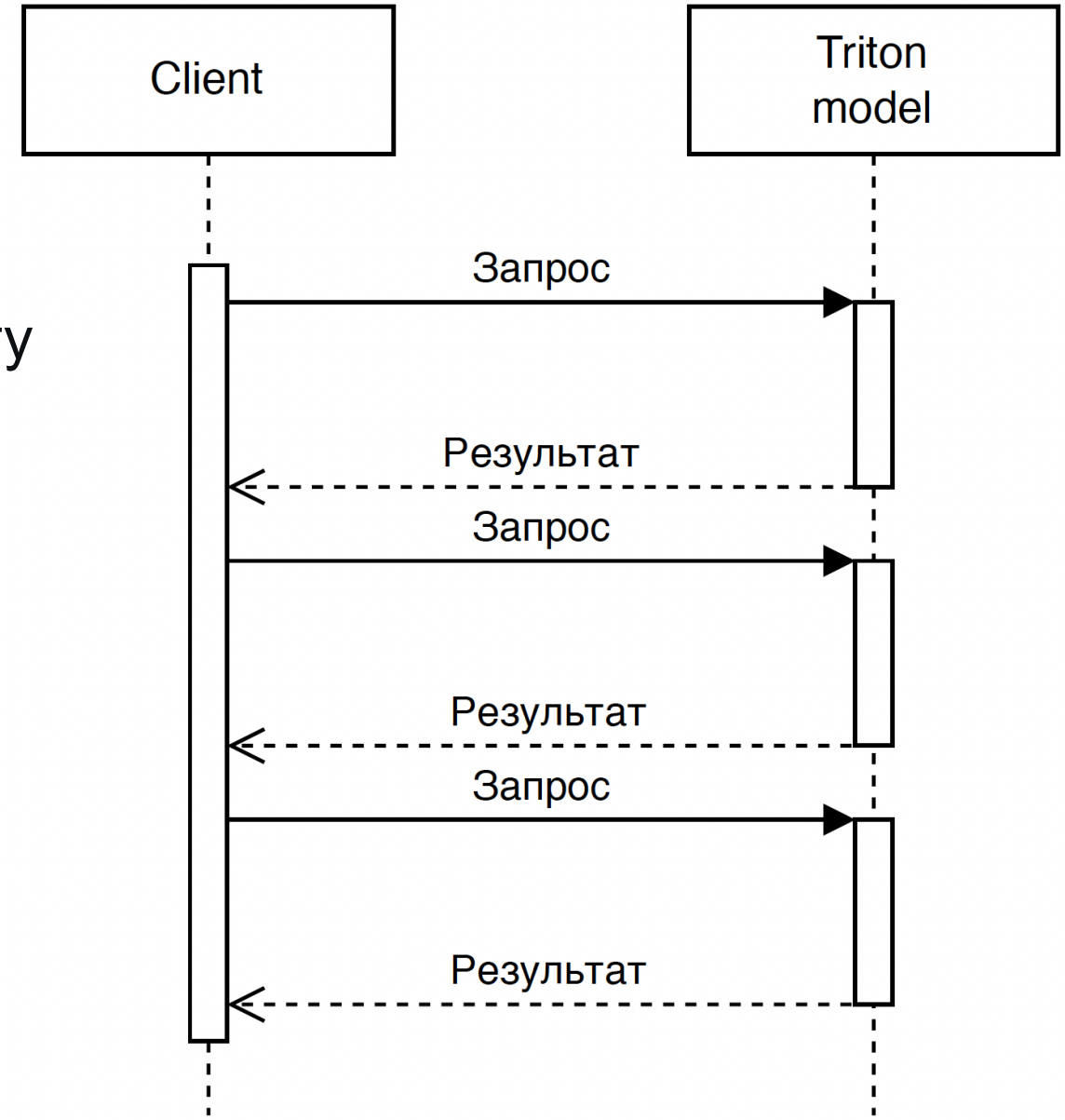
01 { Инстансы



Точка А

- Запросы содержат по одному объекту
- Каждый запрос блокирует модель

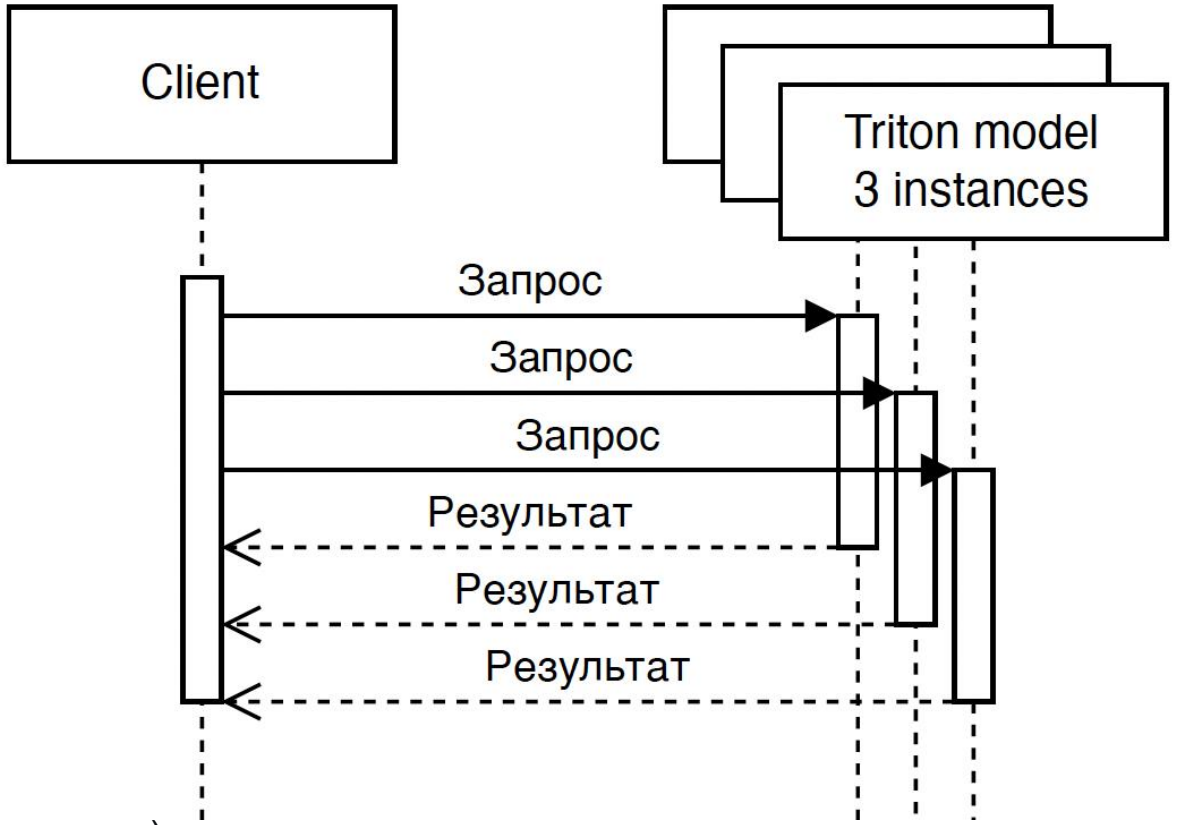
до своего полного выполнения



Наплодим инстансов

Инстанс – ограничение параллелизма.

(execution instance)



Пока первый инстанс обрабатывает первый запрос,
второй запрос может пойти на второй инстанс.

Плюсы:

+ Уменьшение времени простоя запросов
+ Использование большего количества вычислительных ядер

Минусы:

- Дополнительное потребление памяти каждым инстансом (зависит от типа)

  **Python backend ~ 64mb*

  Как выбрать кол-во инстансов

02

Батчевание

Важность батчевания

Оптимизации матричных вычислений

Снижение расходов на вызовы вспомогательных функций

Внешнее батчевание

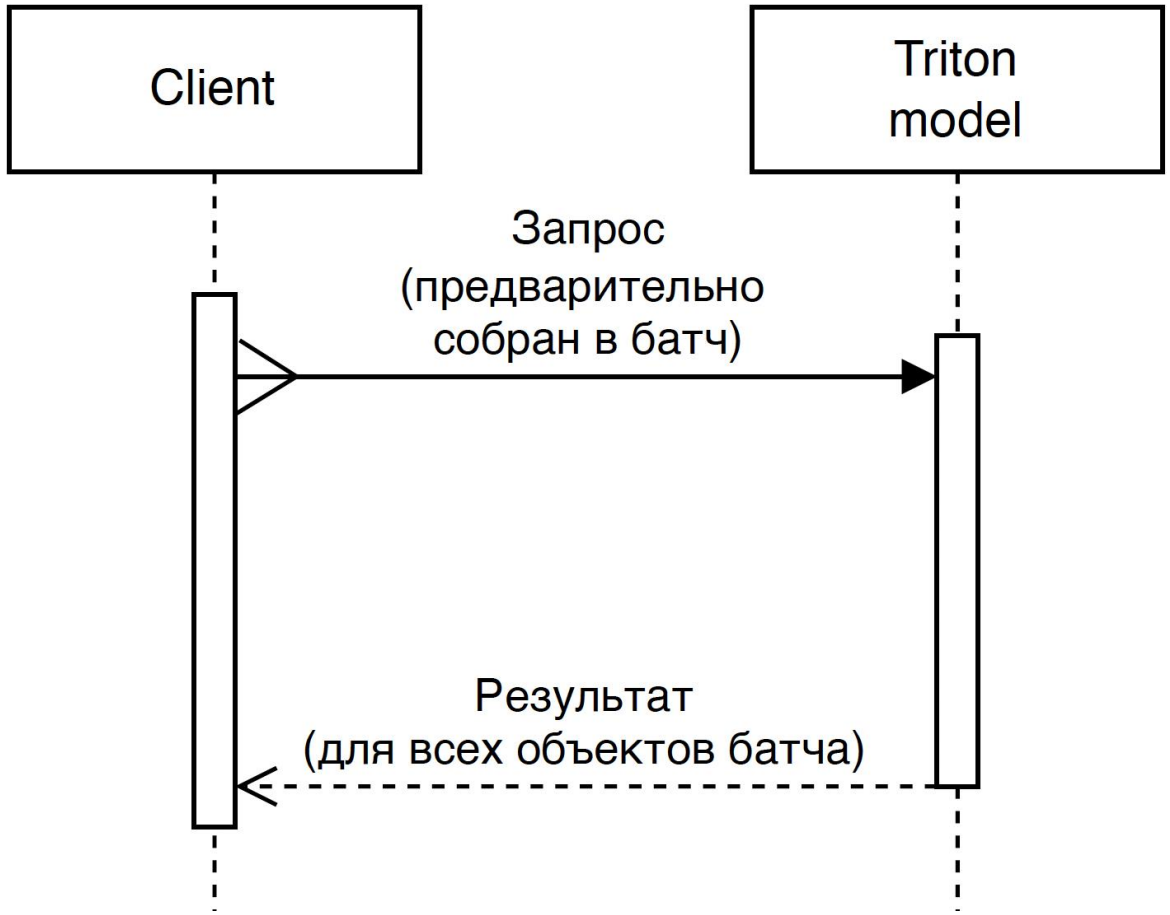
Батч собирается не в тритоне

Плюсы:

+ Все упомянутые преимущества обработки батчами

Минусы:

- Логика сборка батчей переносится в сервис-клиент

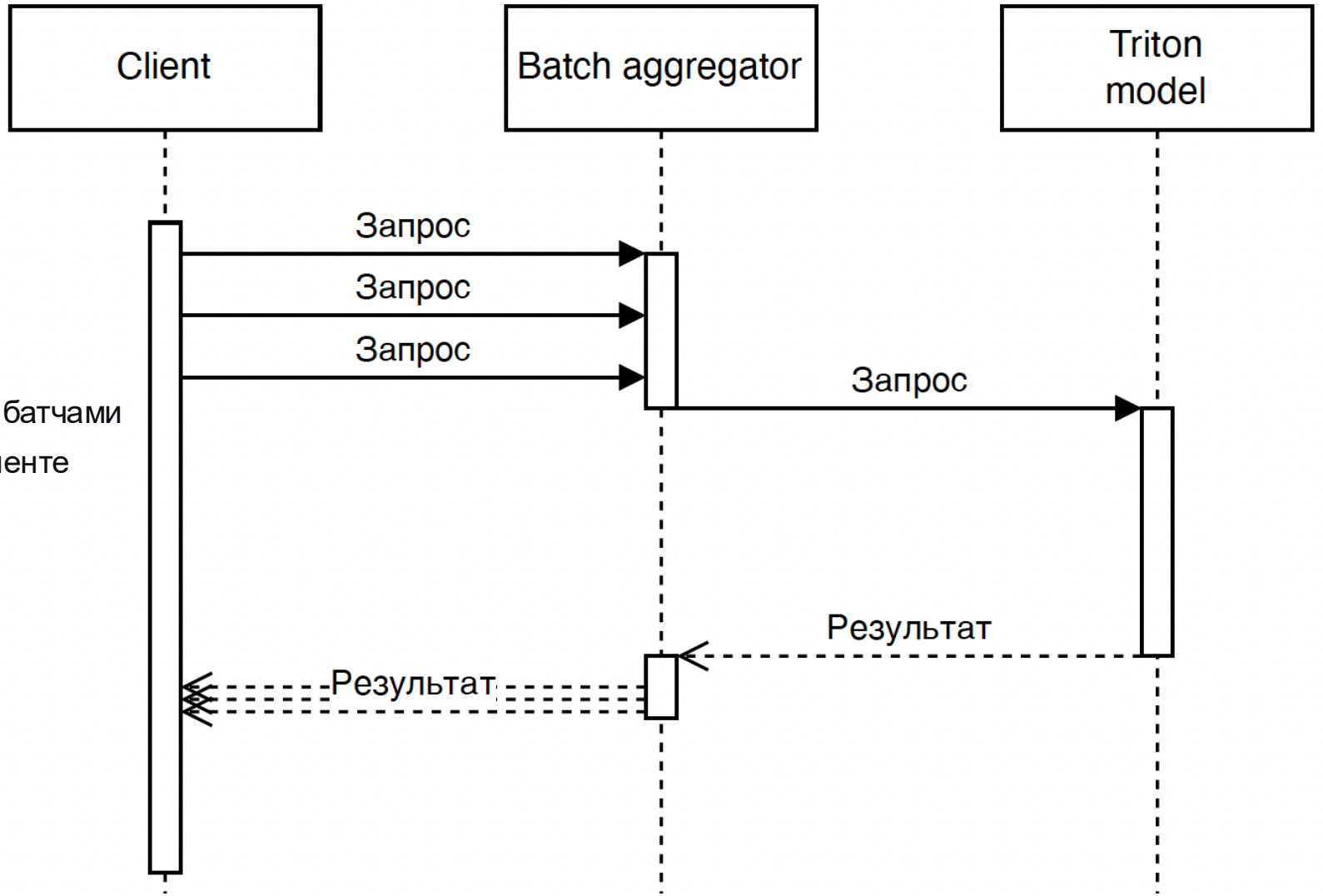


Динамическое батчевание

Батч собирается из накопленных в очереди
или за время ожидания батча элементов

Плюсы:
+ Все упомянутые преимущества обработки батчами
+ Не нужно реализовывать батчевание в клиенте
+ Можно задавать приоритеты запросов

Минусы:

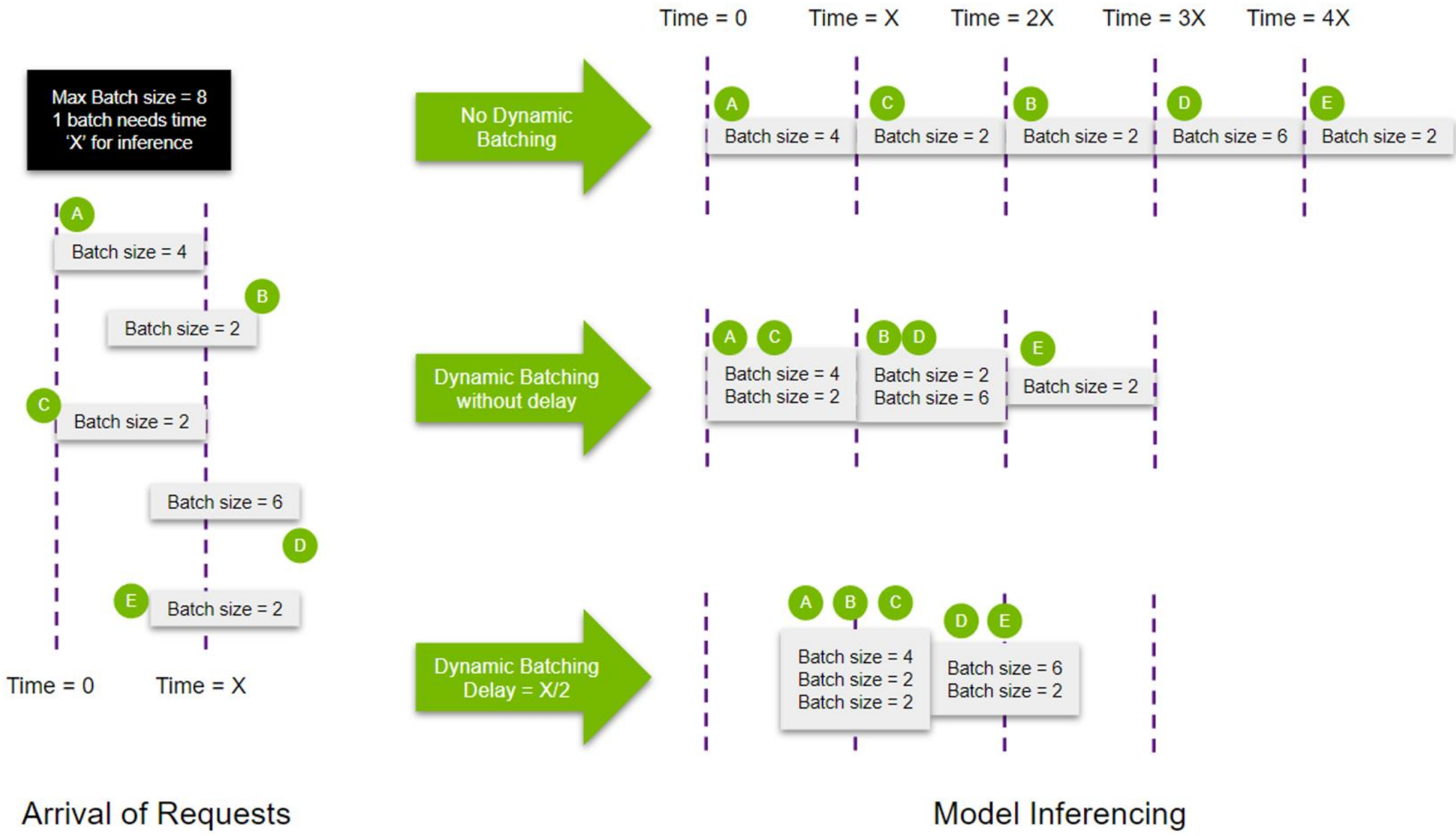


Динамическое батчевание Python backend

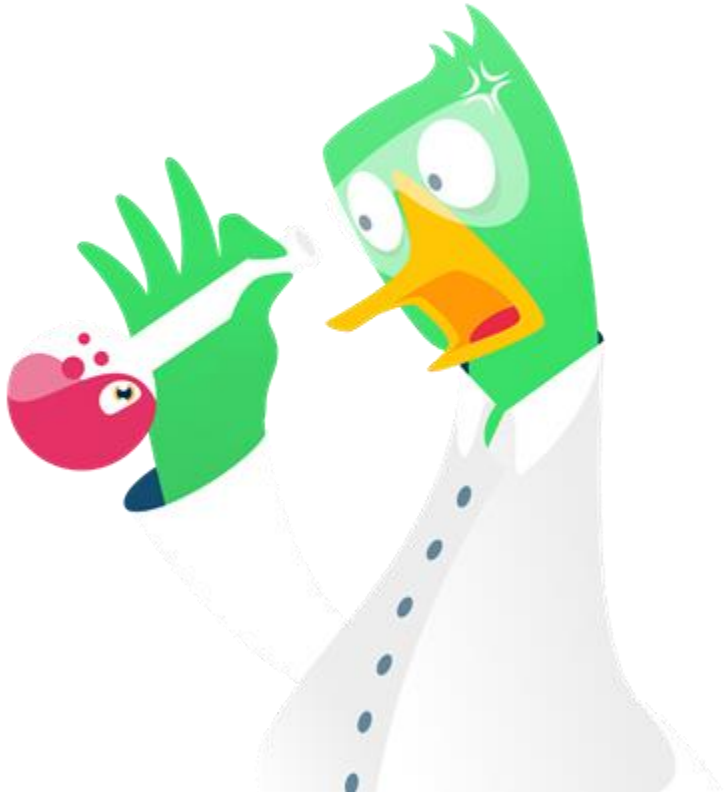
```
def execute(
    self,
    requests: List[pb_utils.InferenceRequest]
) -> List[pb_utils.InferenceResponse]
...

for request in requests:
    ...

    input_tensor = pb_utils.get_input_tensor_by_name(
                    request, "input”
    ).as_numpy()
```


Max Batch size = 8
1 batch needs time 'X' for inference
A
Batch size = 4
B
Batch size = 2
C
Batch size = 2
Batch size = 6
D
E
Batch size = 2
Time = 0
Time = X
Arrival of Requests
No Dynamic Batching
Dynamic Batching without delay
Dynamic Batching Delay = X/2
Time = 0
Time = X
Time = 2X
Time = 3X
Time = 4X
A
Batch size = 4
C
Batch size = 2
B
Batch size = 2
D
Batch size = 6
E
Batch size = 2
A
C
Batch size = 4
Batch size = 2
B
D
Batch size = 2
Batch size = 6
E
Batch size = 2
A
B
C
Batch size = 4
Batch size = 2
Batch size = 2
D
E
Batch size = 6
Batch size = 2
Model Inferencing

03 Оптимизация модели



Интерпретируемый Python класс

Скомпилированная!? Модель

```python
class Llama(GPTBase):
    ...

    def forward(self, idx, targets=None, get_logits=False):
        device = idx.device
        b, t = idx.size()
        assert (
            t <= self.config.sequence_length
        ), f"Cannot forward sequence of length {t}, block size is only {self.config.sequence_length}"
        # shape (1, t)
        pos = torch.arange(0, t, dtype=torch.long, device=device)

        # forward the GPT model itself
        tok_emb = self.transformer.wte(idx)  # token embeddings of shape (b, t, n_embd)

        x = self.transformer.drop(tok_emb)
        freqs_cis = self.freqs_cis.to(x.device)[pos]

        for block in self.transformer.h:
            x = block(x, freqs_cis=freqs_cis)
        x = self.transformer.ln_f(x)

        if targets is not None:
            # if we are given some desired targets also calculate the loss
            logits = self.lm_head(x)
            loss = F.cross_entropy(
                logits.view(-1, logits.size(-1)), targets.view(-1), ignore_index=-1
            )
        else:
            # inference-time mini-optimization: only forward the lm_head on the very last position
            logits = self.lm_head(
                x[:, [-1], :]
            )  # note: using list [-1] to preserve the time dim
            loss = None

        logits = logits if get_logits else None

        return {
            "logits": logits,
            "loss": loss,
        }
```

torch.compile()

Во время первого инференса граф вычислений будет построен, оптимизирован и скомпилирован в c++

Плюсы:

+ Можно использовать по время обучения

Минусы:

- Нет поддержки некоторых операций
- Пока запустить можно только в python бэкенде тритона

```
model = torch.compile(model)
```

Onnx fp16

Простой и надежный вариант

Плюсы:

+ Быстрее чем дефолтный FP32 (не на всех GPU)

Минусы:

- Могут расходиться логиты

```
import onnx
from onnxconverter_common import float16

model = onnx.load("bert.onnx")
model_fp16 = float16.convert_float_to_float16(model)
onnx.save(model_fp16, "bert_fp16.onnx")
```

TensorRT

Плюсы:

+ Работает в тритоне
+ Фьюжен слоев
+ Значительно ускоряет инференс
+ Имеет встроенный профилировщик
  и квантизатор в fp16 и int8

Минусы:

- Требует GPU
- Итоговая модель работает только на той GPU,
  на которой конвертировалась
- После оптимизации могут
  расходиться результаты

```
#Запускать из контейнера с тритоном
/usr/src/tensorrt/bin/trtexec \
 --onnx=/models/onnx_model/1/model.onnx \
 --saveEngine=/models/tensorrt_model/1/model.plan \
 --minShapes=input:1x3x224x224 \
 --optShapes=input:16x3x224x224 \
 --maxShapes=input:32x3x224x224 \
 --int8 \
 --best \
 --verbose \
 --dumpProfile \
 --percentile=95 \
 --exportLayerInfo=/models/trtprofile/LayerInfo.json \
 --exportProfile=/models/trtprofile/profile.json \
 --exportTimes=/models/trtprofile/times.json \
 --separateProfileRun
```

TensorRT – простое решение

Просто добавьте немного этого кода
в конфиг onnx модели …

Плюсы:
+ Не нужно руками конвертировать

Минусы:
- Модель долго инициализируется

```
…
platform: "onnxruntime_onnx"
…
optimization {
 execution_accelerators {
  gpu_execution_accelerator : [
          {
    name : "tensorrt"
    parameters { key: "precision_mode" value: "FP32" }
   }
  ]
 }
}
```

04 { Прочее

Кэширование

Плюсы:

+ Значительное ускорение в неокоторых сценариях
+ Можно подключить отдельный Redis

Минусы:

- Бессмысленно при отсутствии дубликатов в запросах
- Дополнительные затраты памяти на кэш
- Работает только с данными на CPU

```
tritonserver --cache-config local,size=1048576

...

response_cache {
  enable: true
}
```

Клиенты

```
client.async_infer(
    model_name=model_name,
    inputs=inputs,
    outputs=outputs,
    request_id=request_id,
    callback=callback
)
...
async with asyncio.TaskGroup() as tg:
    tasks = [
    tg.create_task(async_infer(
                client, model_name, image)
    )
    for image in images
]
```

```
with concurrent.futures.ThreadPoolExecutor(
    max_workers=num_threads
) as executor:
    futures = [
                executor.submit(
                    infer,
                    client,
                    model_name,
                    image
                )
                for i in range(num_cycles)
    ]
```

Next level

- vLLM
- TensorRT-LLM
- Sequence batching
- Decoupled mode + streaming infer
- Dali

05

Семинар: Запустим перечисленное