Review: HSF Reference Implementation Conditions Database for BelleII
2. Proposed solution

Ruslan Mashinistov, John S. De Stefano Jr, Michel Villanueva

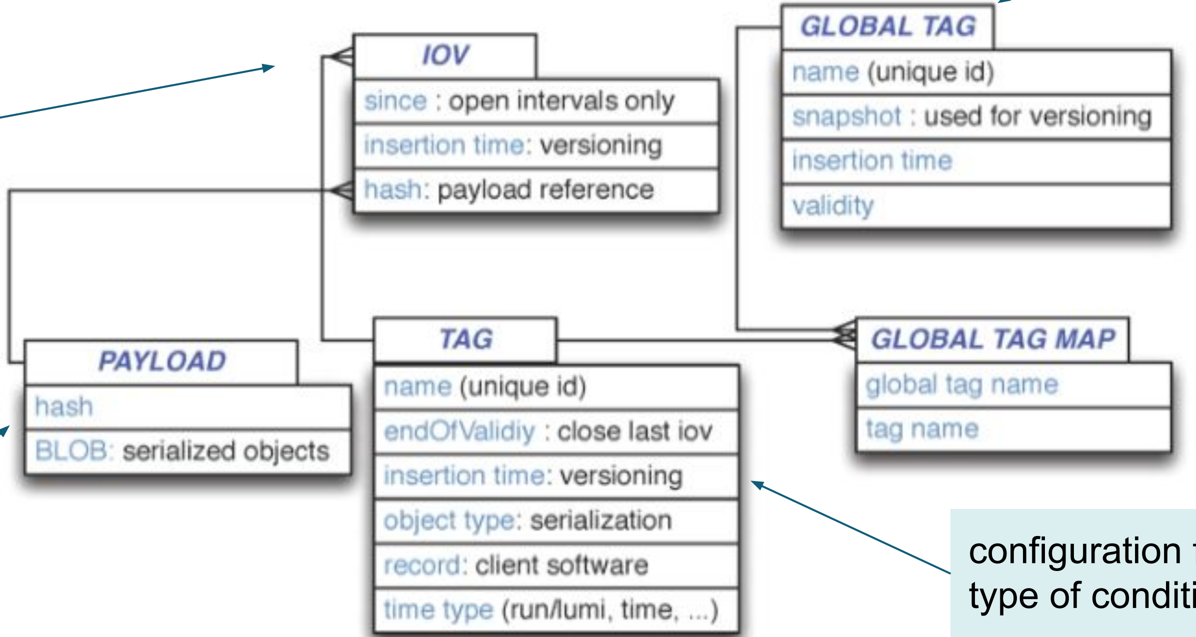
11 July 2024

Belle II HSF CDB Solution Features & Functionality

- Payload agnostic by design, loose server-client coupling (REST Interface)
- Proven scalability O(10M) payloads
- Easy deployment, configuration & horizontal scaling
- Based completely on open source software:
  - Postgres, Django python API, c++ client library
  - Deployed on kubernetes and / or OKD/OpenShift, config via helm
- Integrated support of the common tag workflows
- Various caching options

Conditions Data – HSF Recommendations

- HSF Conditions Databases activity: https://hepsoftwarefoundation.org/activities/conditionsdb.html
  - Discussions across various experiments
- Key recommendations for conditions data handling
  - Separation of payload queries from metadata queries
  - Schema below to organise payloads



IOV
since : open intervals only
insertion time: versioning
hash: payload reference
GLOBAL TAG
name (unique id)
snapshot : used for versioning
insertion time
validity
PAYLOAD
hash
BLOB: serialized objects
TAG
name (unique id)
endOfValidiy : close last iov
insertion time: versioning
object type: serialization
record: client software
time type (run/lumi, time, ...)
GLOBAL TAG MAP
global tag name
tag name

top-level configuration of all conditions data

'Interval of Validity': generalized concept of time (can be time stamp, run number, lumi block, …)

actual data (e.g. in a file)

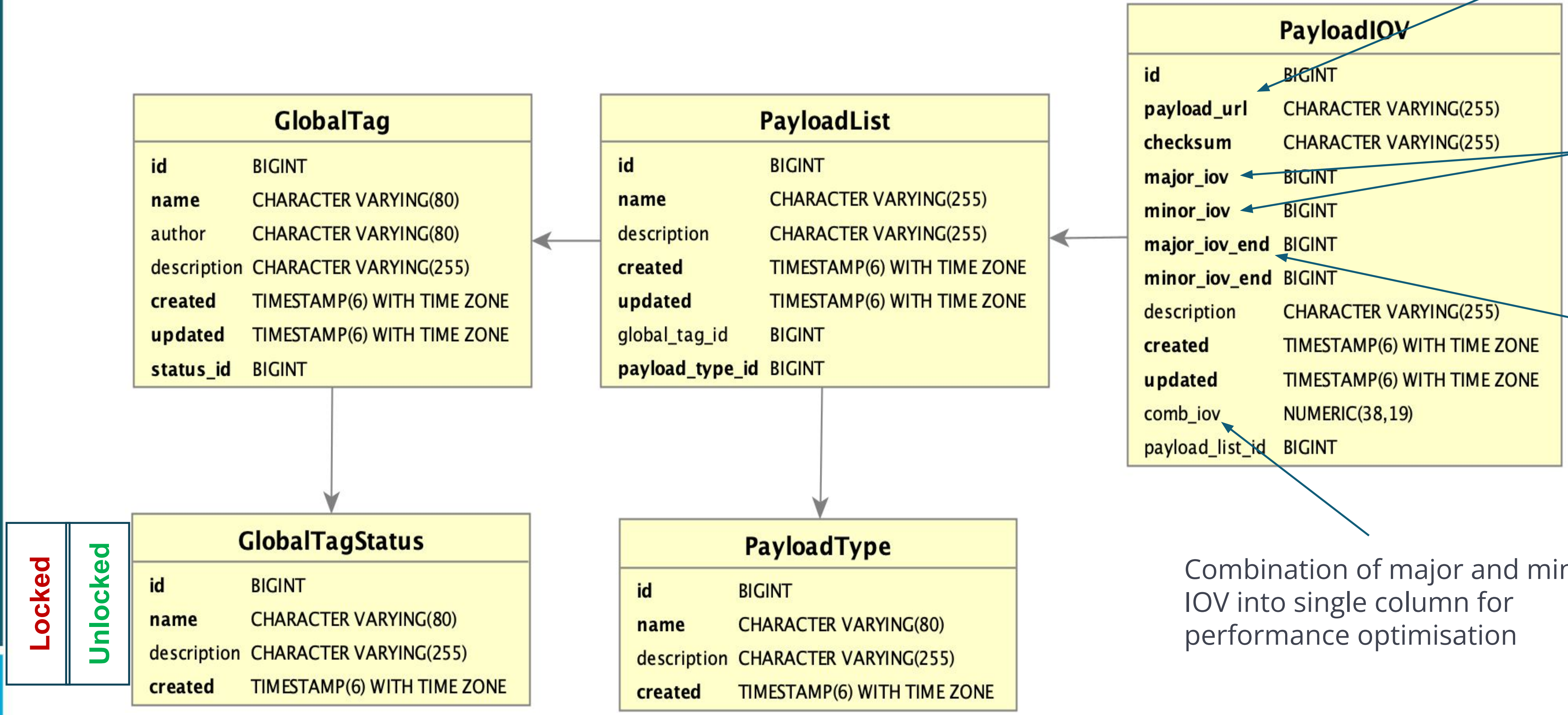
configuration for each type of conditions data

HEP Software Foundation Community White Paper Working Group – Conditions Data

Implementation’s Database schema concept

- **Simple DB Schema**
  - Derived from the HSF Conditions Database white paper
- **Payload objects are combined with IOVs in a single table**
  - Grouped by type and linked to the GT
- **Performance Optimization**
  - Read requests for Payload IOVs are optimized
  - Various techniques and tricks are used to speed up these read requests, as described in the following slides

HSF Database Schema



GlobalTag	
id	BIGINT
name	CHARACTER VARYING(80)
author	CHARACTER VARYING(80)
description	CHARACTER VARYING(255)
created	TIMESTAMP(6) WITH TIME ZONE
updated	TIMESTAMP(6) WITH TIME ZONE
status_id	BIGINT

PayloadList	
id	BIGINT
name	CHARACTER VARYING(255)
description	CHARACTER VARYING(255)
created	TIMESTAMP(6) WITH TIME ZONE
updated	TIMESTAMP(6) WITH TIME ZONE
global_tag_id	BIGINT
payload_type_id	BIGINT

PayloadIOV	
id	BIGINT
payload_url	CHARACTER VARYING(255)
checksum	CHARACTER VARYING(255)
major_iov	BIGINT
minor_iov	BIGINT
major_iov_end	BIGINT
minor_iov_end	BIGINT
description	CHARACTER VARYING(255)
created	TIMESTAMP(6) WITH TIME ZONE
updated	TIMESTAMP(6) WITH TIME ZONE
comb_iov	NUMERIC(38,19)
payload_list_id	BIGINT

GlobalTagStatus	
id	BIGINT
name	CHARACTER VARYING(80)
description	CHARACTER VARYING(255)
created	TIMESTAMP(6) WITH TIME ZONE

PayloadType	
id	BIGINT
name	CHARACTER VARYING(80)
description	CHARACTER VARYING(255)
created	TIMESTAMP(6) WITH TIME ZONE

Payloads are not stored in schema

major- & minor IOV for more flexibility

IOVs also have an end

Combination of major and minor IOV into single column for performance optimisation

Combined IOV column

- Querying by two IOVs, namely major and minor (experiment and run numbers in Belle II), can be a bit complex. To streamline this process, we've implemented a combined IOV field. Here's how it works:
- **Combined IOV Field**:
    - Integrates both major and minor IOVs
        - **Major IOV**: Represents the whole part.
        - **Minor IOV**: Represents the fractional part.
    - Used internally to speed up queries.
    - Hidden from users for simplicity.

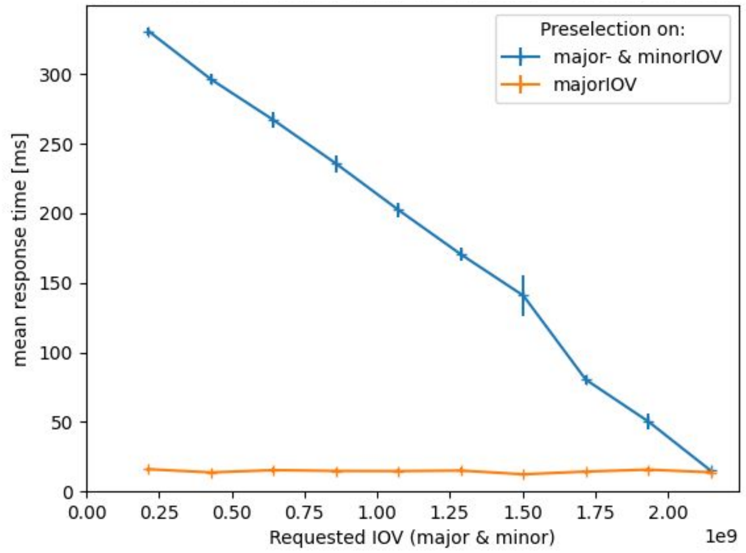
```
comb_iov = models.DecimalField(db_column='comb_iov', max_digits=38, decimal_places=19, null=True)

data['comb_iov'] = Decimal(Decimal(data["major_iov"]) + Decimal(data["minor_iov"]) / 10 ** 19)
```

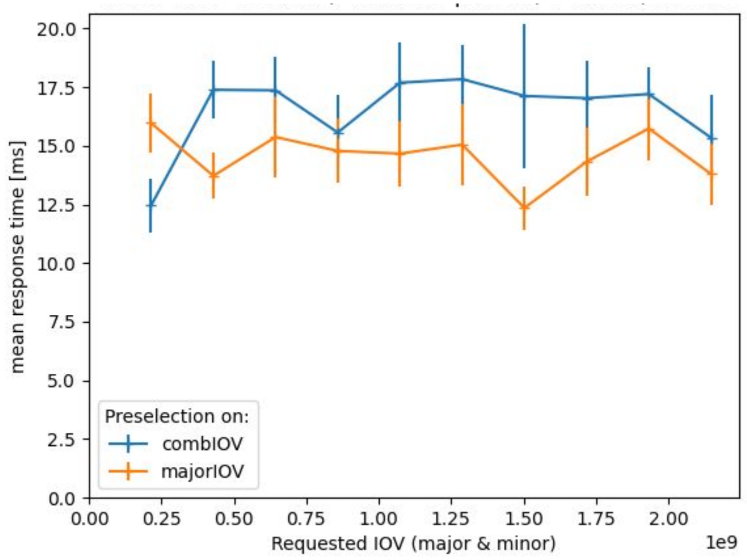
Covering index

- **<u>Definition and Benefits</u>**:
  - A covering index in PostgreSQL is an index that includes all the columns required to satisfy a query, allowing the query to be executed entirely from the index without accessing the table. This can significantly improve query performance by reducing I/O operations.
  - Covering indexes are particularly useful for read-heavy operations where specific queries are frequently executed. They help in reducing the number of data pages read from the disk, thus speeding up query execution times.

```
class Meta:
    db_table = u'PayloadIOV'
    indexes = [
        models.Index('payload_list', F('comb_iov').desc(nulls_last=True), name='covering_idx')
        ]
```

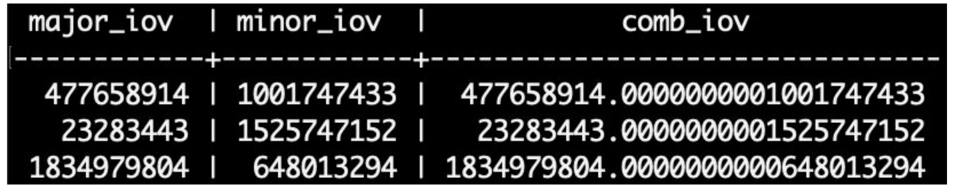
Raw SQL - Combined IOV Column





- Preselection on major- & minor IOV (AND / OR)
  - Scales with entries to consider
  - Query uses ‘Filter’
- Preselection on single column (<=)
  - Constant time
  - Query uses ‘Index Condition’

- Combine major- and minor IOV into single column:



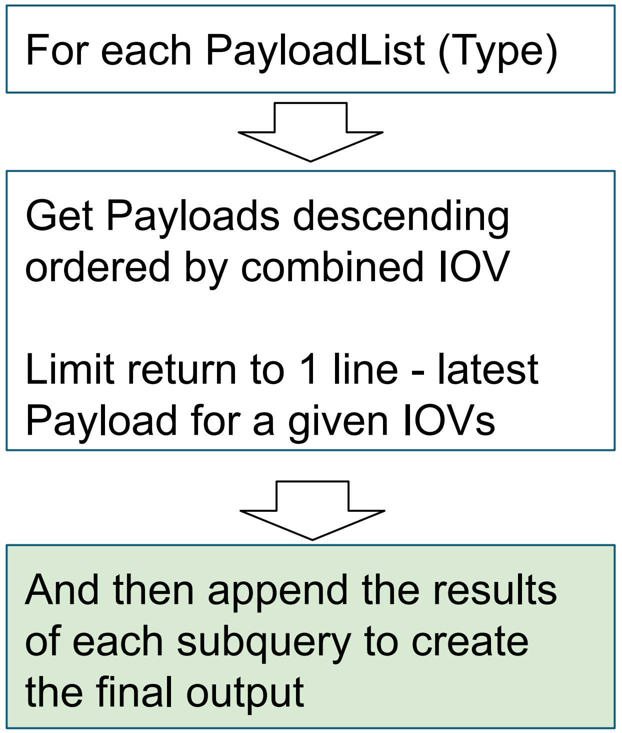
```
 major_iov  | minor_iov  |             comb_iov
------------+------------+--------------------------------
  477658914 | 1001747433 |  477658914.0000000001001747433
   23283443 | 1525747152 |   23283443.0000000001525747152
 1834979804 |  648013294 | 1834979804.0000000000648013294
```

bigint bigint decimal(38, 19)

- Fast across all values while selecting on both

PayloadIOV Read API – Raw SQL Query

```sql
SELECT pi.payload_url, pi.major_iov, pi.minor_iov,
pt.name, …
FROM "PayloadList" pl
JOIN "GlobalTag" gt ON pl.global_tag_id = gt.id AND
gt.name = %(my_gt)s
JOIN LATERAL (
  SELECT payload_url, major_iov, minor_iov, …
  FROM   "PayloadIOV" pi
  WHERE  pi.payload_list_id = pl.id
    AND pi.comb_iov <= CAST(%(my_major_iov)s +
CAST(%(my_minor_iov)s AS DECIMAL(19,0)) / 10E18 AS
DECIMAL(38,19))
  ORDER BY pi.comb_iov DESC
  LIMIT 1
) pi ON true
JOIN "PayloadType" pt ON pl.payload_type_id = pt.id;
```



For each PayloadList (Type)

Get Payloads descending ordered by combined IOV

Limit return to 1 line - latest Payload for a given IOVs

And then append the results of each subquery to create the final output

- LATERAL joining. Without LATERAL, each sub-SELECT is evaluated independently and so cannot cross-reference any other FROM item
- Covering index on Payload table including combined IOV and reference to the PayloadList

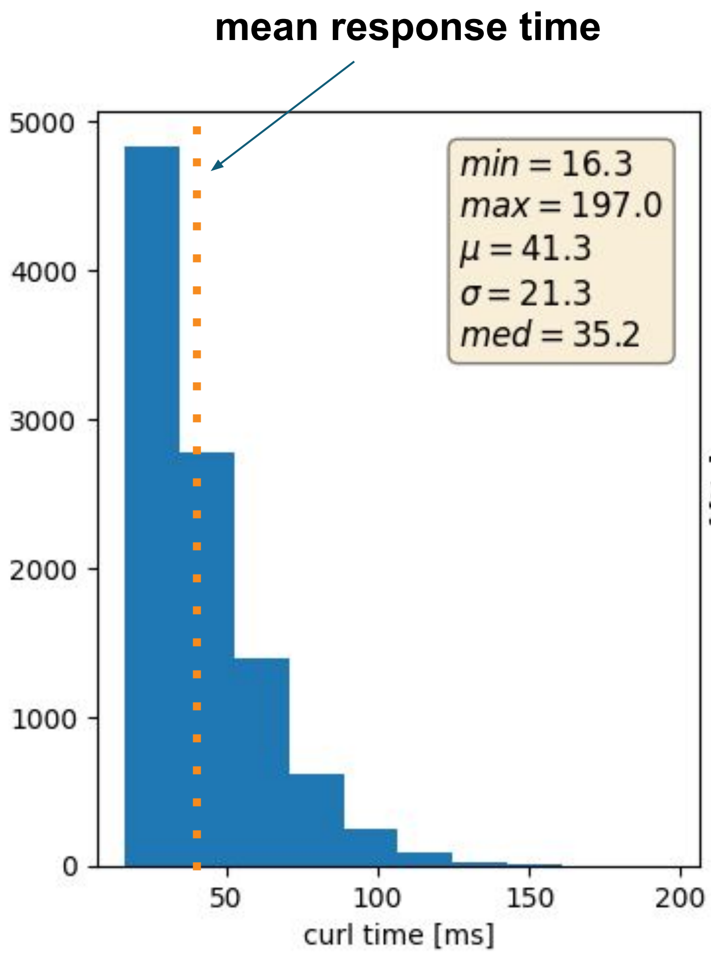
Performance Testing – Strategy

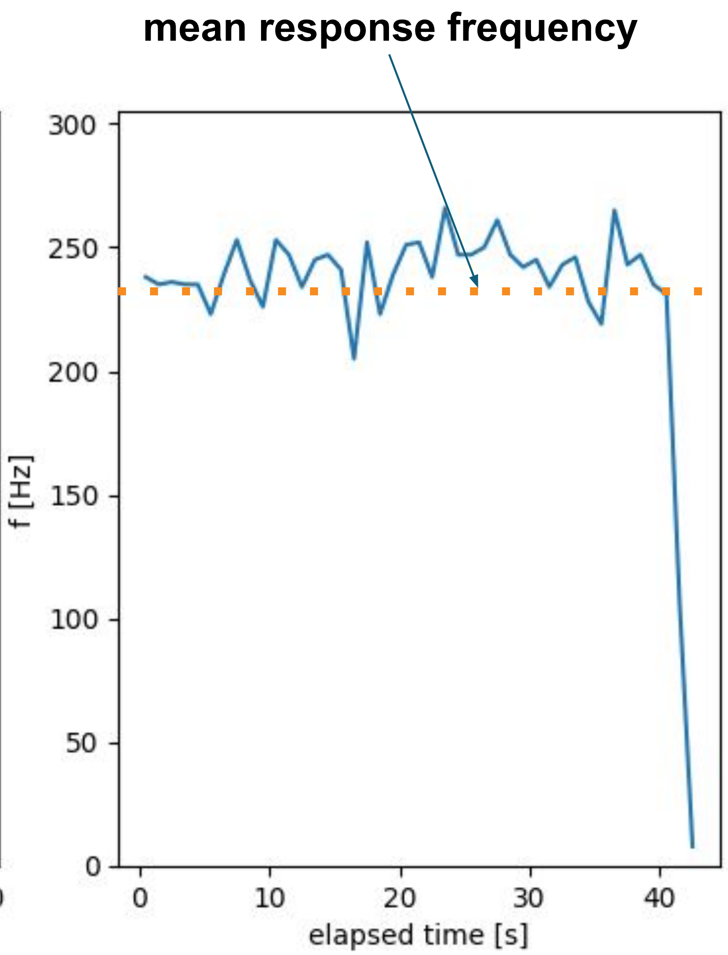
- Simulate expected DB occupancy
- Simulate access patterns
  - Query read API for payload URL
  - Parallel requests via HTC or MT

Scenario	Payload Types	Payload IOVs (per type)
tiny	10	100 (10)
tiny-moderate	10	2000 (200)
moderate	100	20000 (200)
heavy-usage	100	500000 (5000)
worst-case	200	5200000 (26000)

All following tests:

- Random major- and minor IOV, no caching
- Query metadata only, no payloads

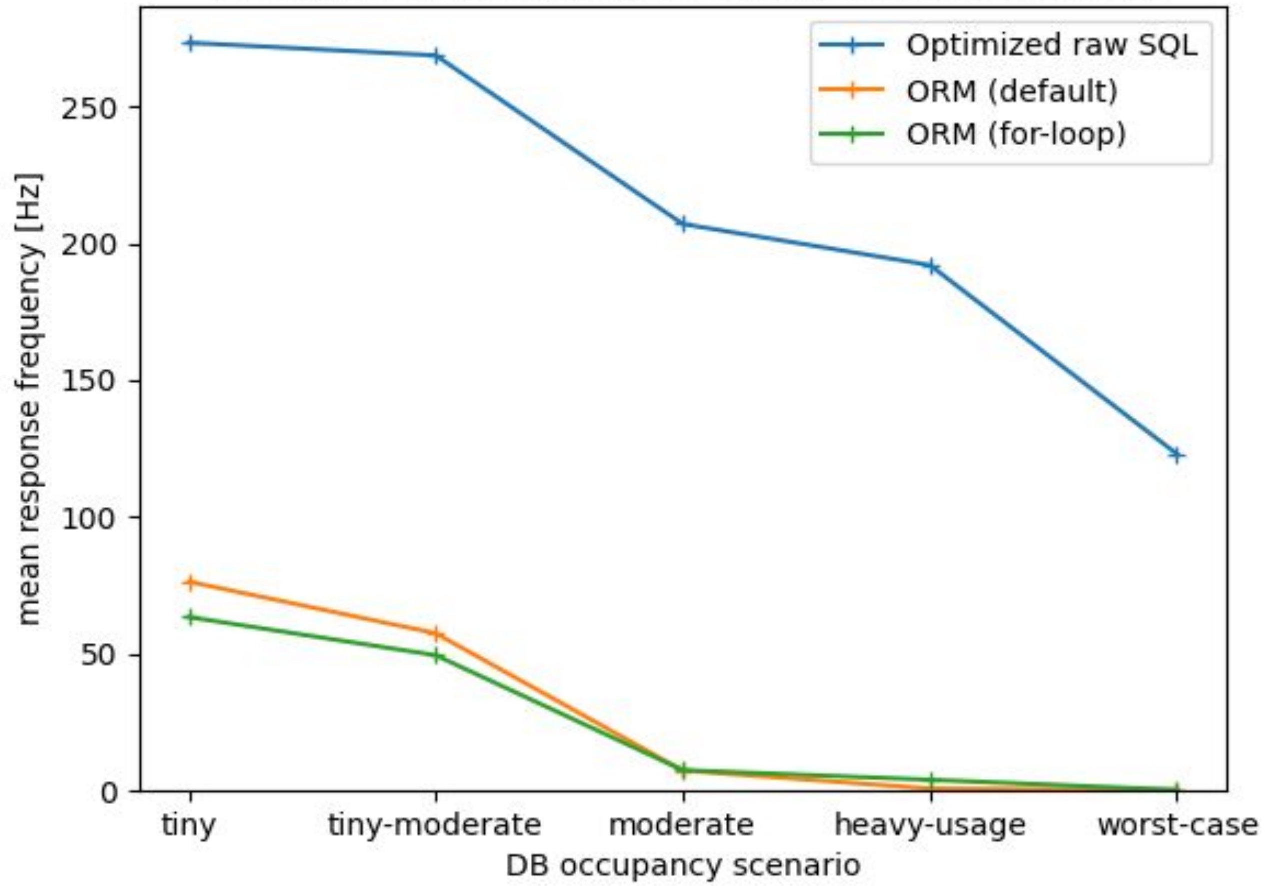




Performance Testing – ORM vs Raw SQL

- High frequency read API workflow:
  - Filter on global tag, major- and minor IOV *
  - Find 'latest' IOV for each payload type **
  - Return payload type, file URL, IOV
- Django's ORM writes query for user
- Optimized raw SQL query
  - Covering index (index-only scan)
  - Combined IOV column <major.minor>
  - Lateral join operation

Resp. freq. vs size of queried GT

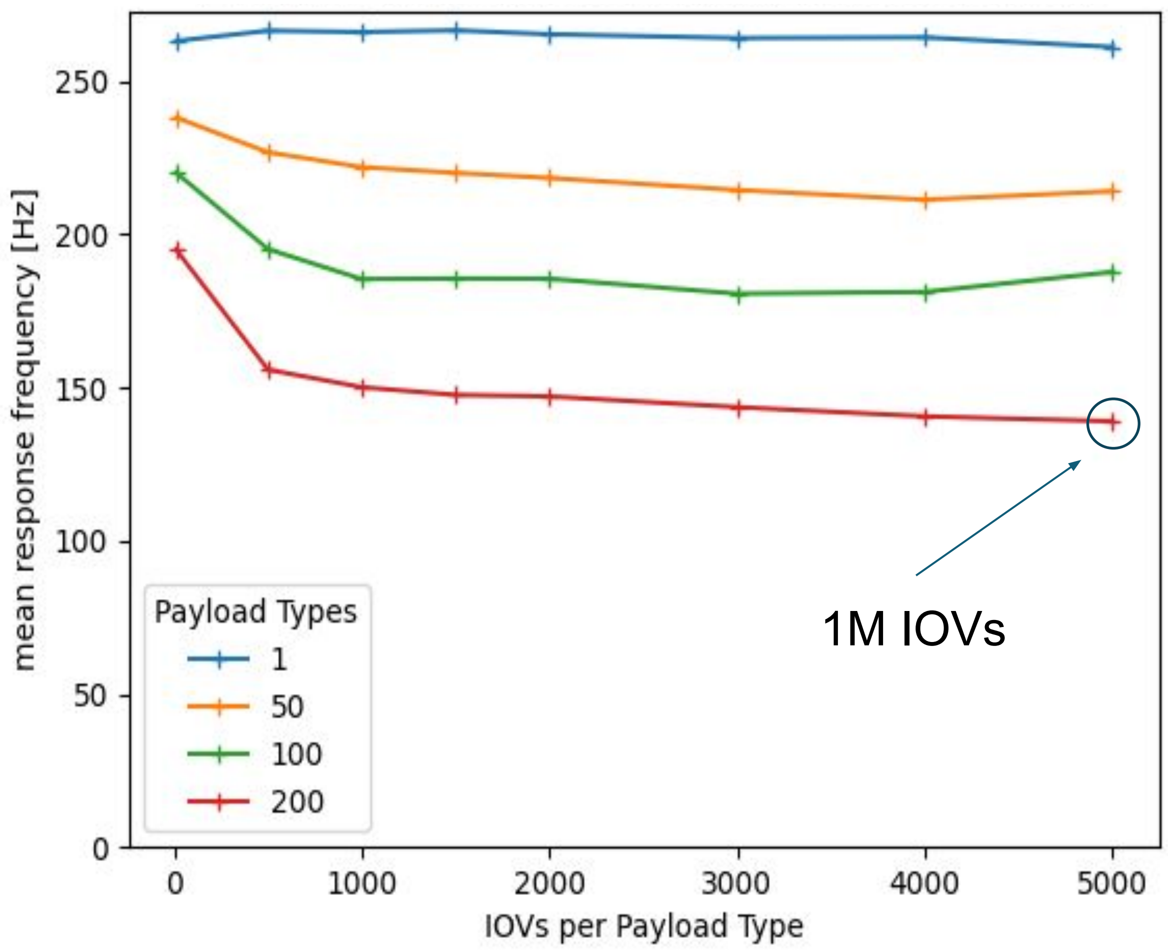


*: my_major<major_iov OR (my_major=major_iov AND my_minor<=minor_iov) **: for max major_iov, find max minor_iov

Performance Testing – Scaling

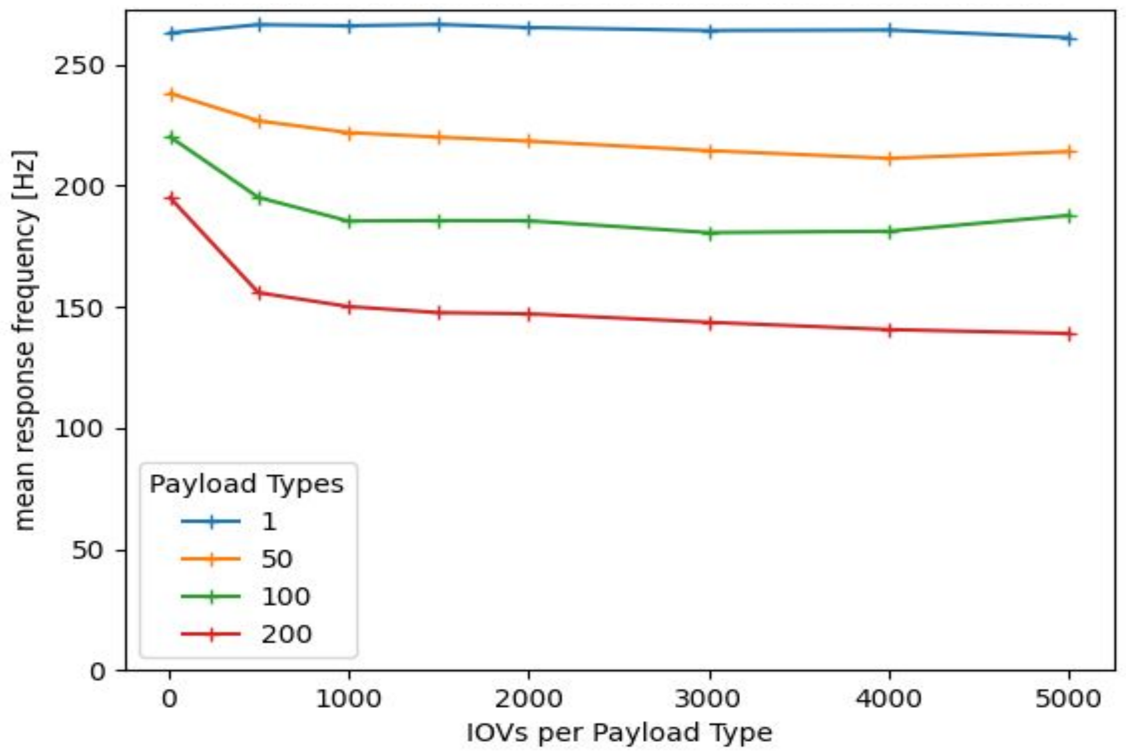
- Investigate scaling w/ size of queried GT
  - Content of DB remains constant
- Measure mean response frequencies
  - Scales with number of payload types
    - More data to sort and return
  - Almost flat vs number of IOVs
    - Index scan (covering index)
- Also tested scaling w.r.t. size of DB
  - No dependence, plot in backup
- Other tricks used to reach this performance:
  - Combined IOV
  - Lateral joining
- Cloning of the GT of 100K payloadIOVs takes only ~30 sec

Resp. freq. vs size of queried GT

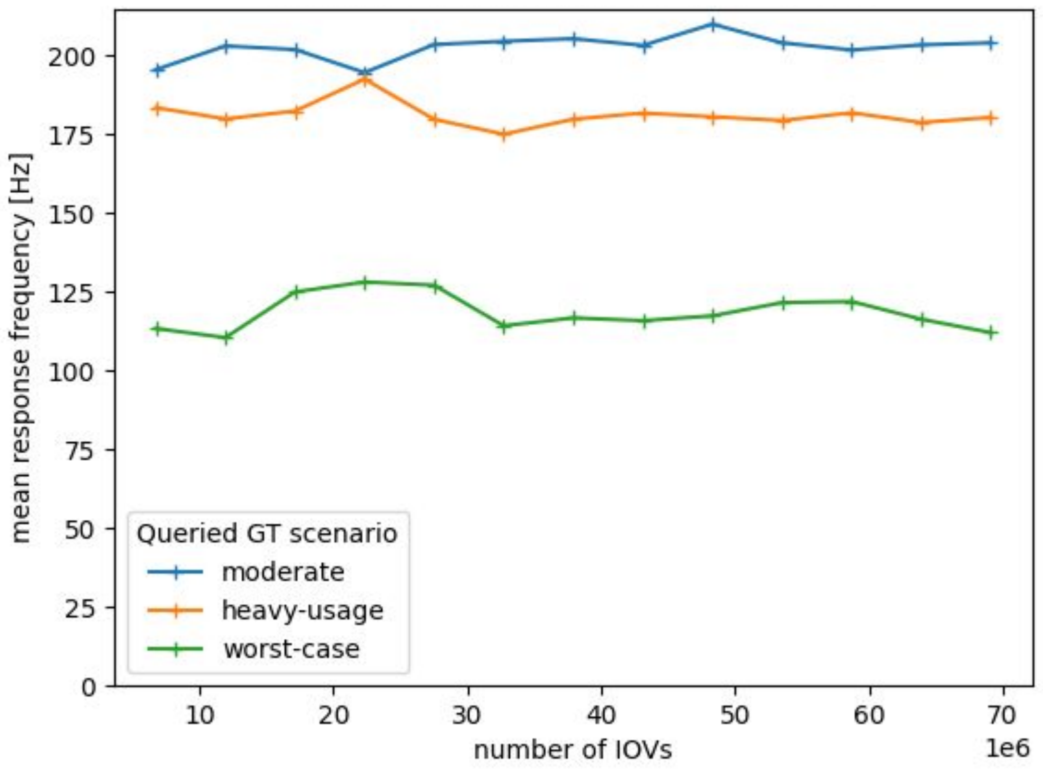


Performance Testing – Scaling

Resp. freq. vs size of queried GT



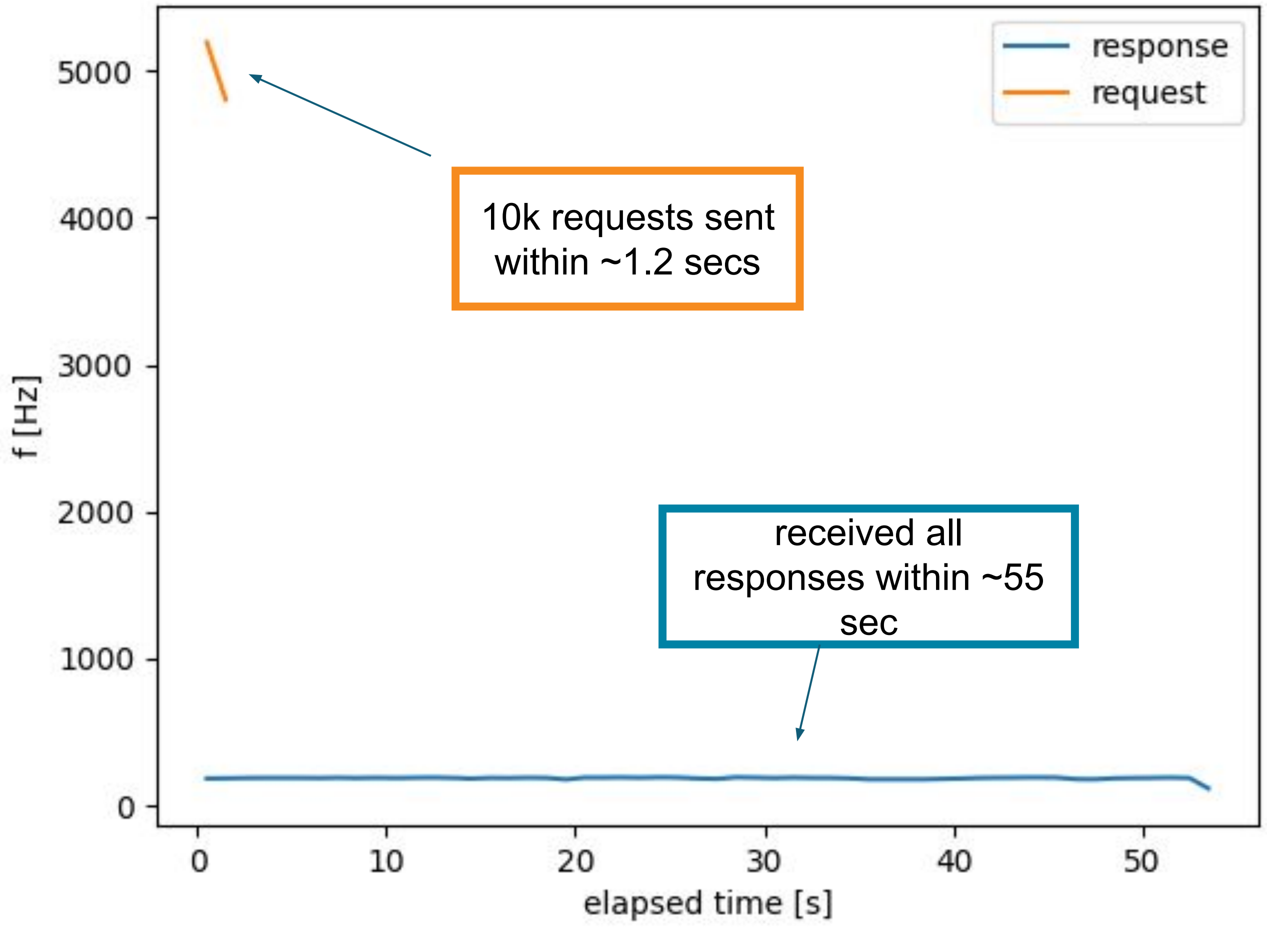
Resp. freq. vs DB size



- Scales with number of payload types
- Almost flat w.r.t. number of IOVs

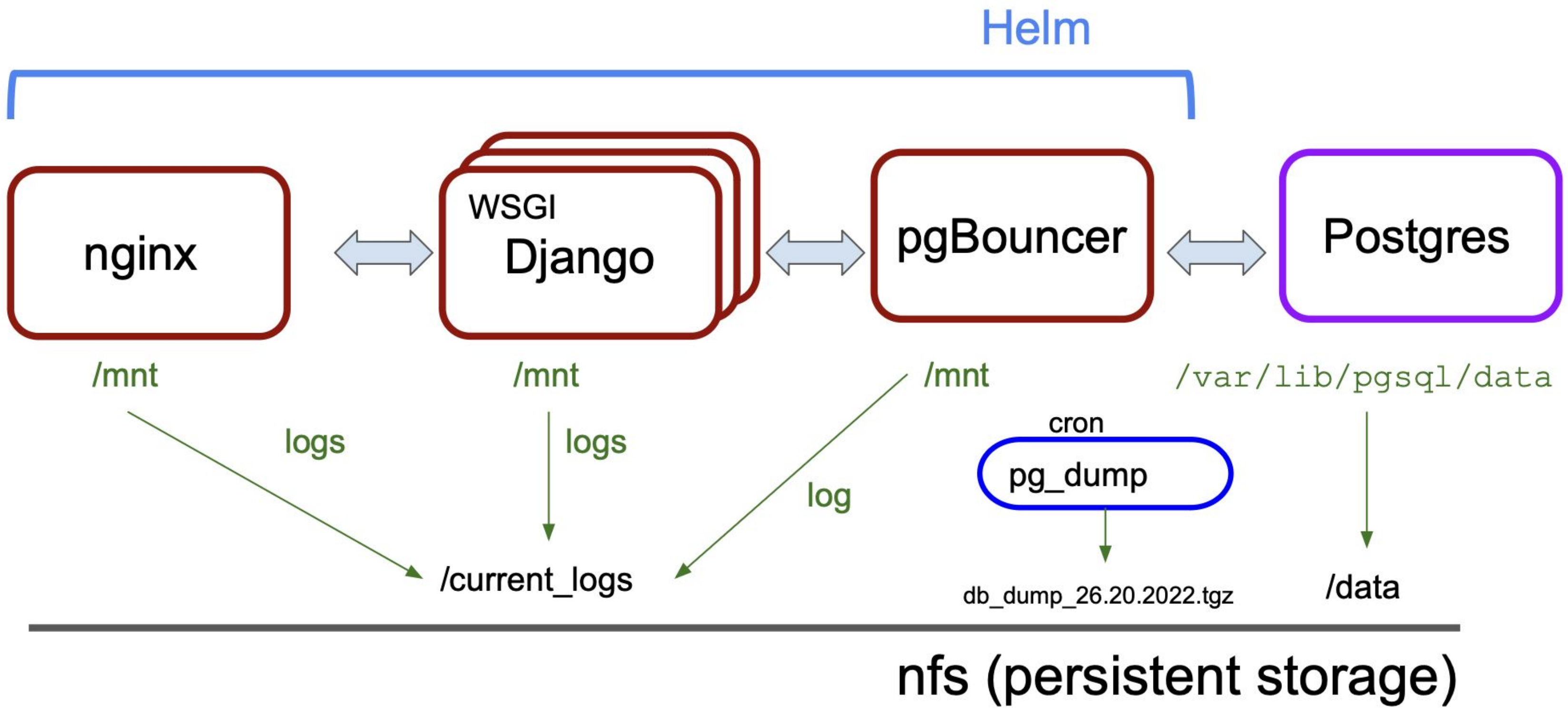
- Performance depends on size of queried GT
- Additional ‘stuff’ in DB has no significant impact

Performance Testing – High Frequency



- Simulate offline reco use case
  - Many jobs launched at same time
- Cooperative multithreading (**asynchio**)
  - Send requests firsts
  - Process responses later
- Allows very high peak request frequency
- Server-side queuing of requests works

Deployment on OKD (OpenShift)



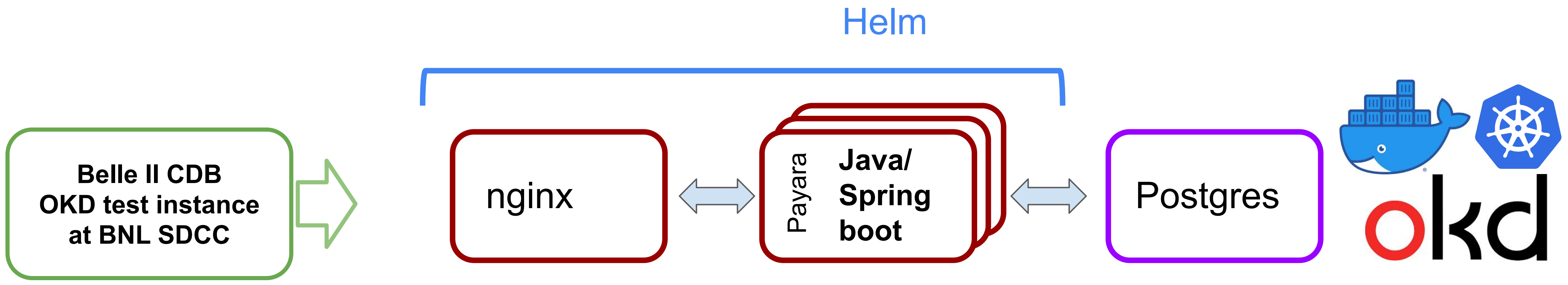
- Helm streamlines the deployment of Kubernetes clusters
- Classic deployment at VMs also possible and has been tested
- all-in-a-single-container image available

- Automated deployment on OKD (OpenShift) using Helm chart
- Horizontally scalable
- Open Source only

Easily adoptable for various HEP experiments

Belle II migration to OKD/OpenShift

- Due to issues with our existing Kubernetes infrastructure, we have initiated a migration to OKD/OpenShift
  - We are adapting the HSF Helm deployment configuration to support our current Java application
  - We have already successfully conducted a series of functional tests
  - This progress will significantly streamline future migration

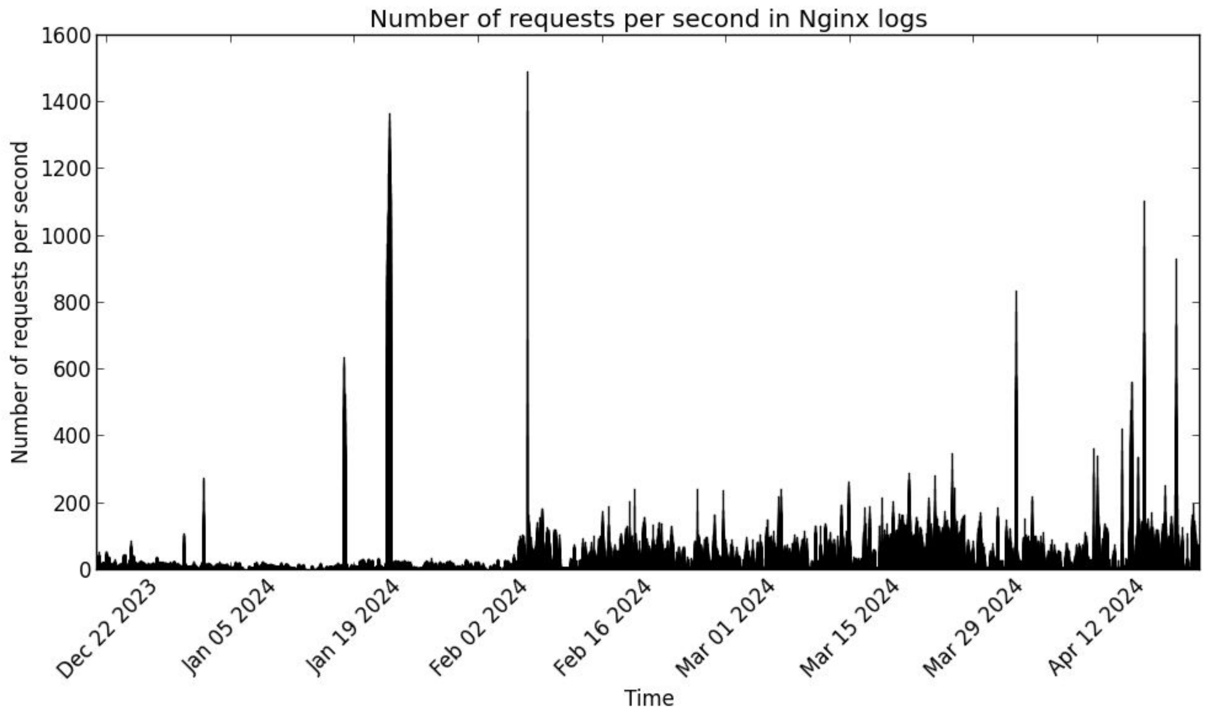


This progress will significantly streamline the complete future migration
This shouldn't affect current operation

One Year of Successful Production in sPHENIX

<u>Valuable Experience gathered</u>:

- Bugfixes regarding retry mechanism, check for file system write permission,
  and sPHENIX's compiler optimizations
- CDB throughput issue at the level of ~20K almost concurrent jobs.
  - Implemented very conservative Nginx caching: 1sec for most used resource call
  - Future plan: different client-configurable server-side caching strategies



Experience from DUNE

- **nopayloadclient** has been accepted into SciSoft (FNAL)
- Created prototype for DUNE-specific client: **dunenpc**
  - Developed **art** *Service* to interface **dunenpc**
- Deployed test instance of backend @ CERN
  - Apache & bare Django on VM (for integration tests)
  - Created corresponding configuration file

→ Successfully ran DUNE offline dummy job w/ access to our DB

Conclusion

- We have observed an increase in problems and issues with the Belle II CDB
    - Additionally, we anticipate significant limitations with the current implementation
- We're considering HSF CDB as the candidate to replace current service
    - Django REST API: nopayloaddb
        - Automated deployment on OKD with helm-chart
    - C++ client-side client: nopayloadclient
- HSF CDB performance tests show solid results
- One year of successful production in sPHENIX
    - Also experience is gained from the Dune test-instance
    - ePIC collaboration is now considering the migration to the HSF CDB

Backup

Brookhaven National Laboratory

Conditions Data – Introduction

“Conditions data is any additional data needed to process event data”

Changes over time

- Repeat detector calibration with larger cosmic dataset
- Improve calibration algorithms

Versioning & configuration

High access rates

- Distributed computing jobs access same conditions data simultaneously
- Access rates up to ~kHz

Fast DB queries & effective caching

Heterogeneous data

- Granularity varies (time indexed, run-indexed, constant)
- Structure of payload varies (3D map, time-indexed values, single number, …)

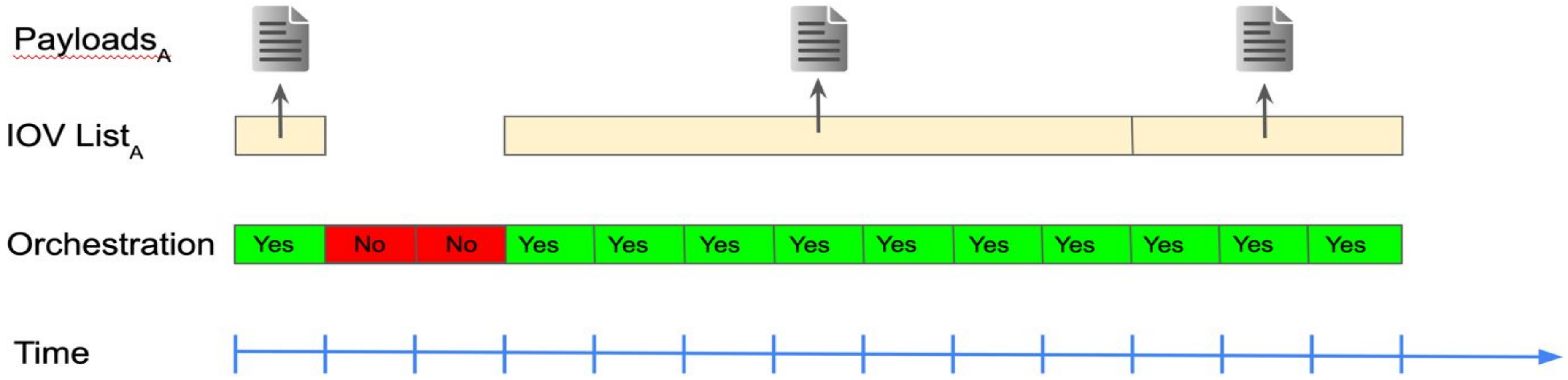
Payload agnostic by design

Similar challenges for various HEP experiments

Conditions Data – Use Cases

- HSF Conditions Database meeting: **use cases** https://indico.cern.ch/event/1280790/
- Most can be realised w/ HSF Recomm.
- Most demanding use-case is **Fast-Processing**. Goal:
  - Publish data for analysis fast
  - Maximize physics performance

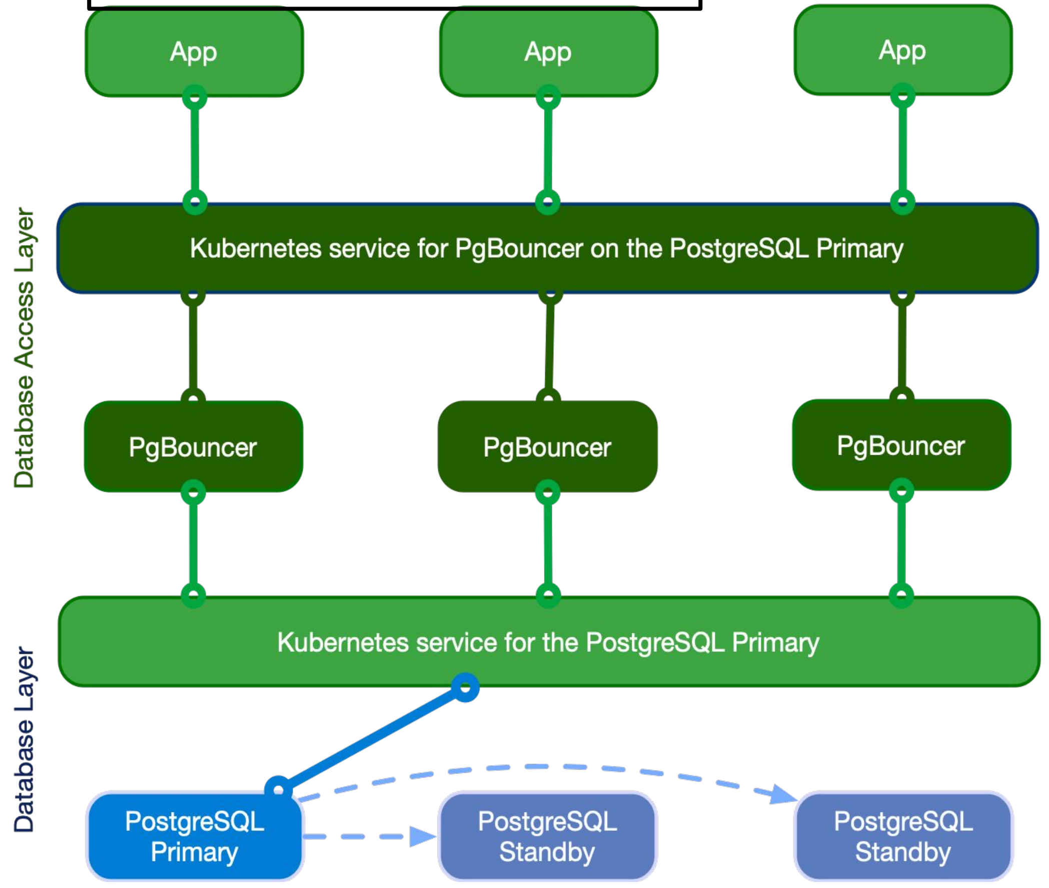
Use case	Example
Online	• High Level Trigger
Reprocessing	• Run reco w/ improved calib.
Analysis	• High level physics analysis
Development	• Test new calib. within existing GT
★ Fast-processing	• Process data w/ just-in-time calib.



PostgreSQL High-Availability Cluster

- Consider DB cluster for high-availability and higher performance
- CloudNativePG:
  - Open source operator (Kubernetes) for PostgreSQL
  - Primary / Standby architecture
  - Native support for pgBouncer connection pooling

Open source Kubernetes operator for HA PostgreSQL



Investigating Query Plans - I

```
Hash Join  (cost=7.23..410.15 rows=86 width=70) (actual time=6.111..365.158 rows=200 loops=1)
  Hash Cond: (pl.payload_type_id = pt.id)
  ->  Nested Loop  (cost=0.71..403.40 rows=86 width=69) (actual time=6.017..364.977 rows=200 loops=1)
        ->  Nested Loop  (cost=0.15..11.70 rows=86 width=16) (actual time=0.048..0.133 rows=201 loops=1)
              ->  Seq Scan on "GlobalTag" gt  (cost=0.00..1.09 rows=1 width=8) (actual time=0.023..0.025 rows=1 loops=1)
                    Filter: ((name)::text = 'worst-case'::text)
                    Rows Removed by Filter: 6
              ->  Index Scan using "PayloadList_global_tag_id_2b35c85f" on "PayloadList" pl
                        (cost=0.15..9.75 rows=86 width=24) (actual time=0.022..0.083 rows=201 loops=1)
                    Index Cond: (global_tag_id = gt.id)
        ->  Limit  (cost=0.56..4.53 rows=1 width=61) (actual time=1.815..1.815 rows=1 loops=201)
              ->  Index Only Scan using combo_covering_idx on "PayloadIOV" pi
                        (cost=0.56..3484.55 rows=876 width=61) (actual time=1.815..1.815 rows=1 loops=201)
                    Index Cond: (payload_list_id = pl.id)
                    Filter: ((major_iov < 100000000) OR ((major_iov = 100000000) AND (minor_iov <= 100000000)))
                    Rows Removed by Filter: 24669
                    Heap Fetches: 0
  ->  Hash  (cost=4.01..4.01 rows=201 width=17) (actual time=0.078..0.078 rows=201 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 19kB
        ->  Seq Scan on "PayloadType" pt  (cost=0.00..4.01 rows=201 width=17) (actual time=0.018..0.043 rows=201 loops=1)
Planning Time: 0.996 ms
Execution Time: 365.221 ms
```

major- & minorIOV

```
Hash Join  (cost=7.23..90.89 rows=86 width=70) (actual time=0.309..3.244 rows=200 loops=1)
  Hash Cond: (pl.payload_type_id = pt.id)
  ->  Nested Loop  (cost=0.71..84.14 rows=86 width=69) (actual time=0.075..2.935 rows=200 loops=1)
        ->  Nested Loop  (cost=0.15..11.70 rows=86 width=16) (actual time=0.028..0.121 rows=201 loops=1)
              ->  Seq Scan on "GlobalTag" gt  (cost=0.00..1.09 rows=1 width=8) (actual time=0.013..0.018 rows=1 loops=1)
                    Filter: ((name)::text = 'worst-case'::text)
                    Rows Removed by Filter: 6
              ->  Index Scan using "PayloadList_global_tag_id_2b35c85f" on "PayloadList" pl
                        (cost=0.15..9.75 rows=86 width=24) (actual time=0.012..0.063 rows=201 loops=1)
                    Index Cond: (global_tag_id = gt.id)
        ->  Limit  (cost=0.56..0.82 rows=1 width=61) (actual time=0.014..0.014 rows=1 loops=201)
              ->  Index Only Scan using combo_covering_idx on "PayloadIOV" pi
                        (cost=0.56..232.55 rows=876 width=61) (actual time=0.013..0.013 rows=1 loops=201)
                    Index Cond: ((payload_list_id = pl.id) AND (major_iov < 100000000))
                    Heap Fetches: 0
  ->  Hash  (cost=4.01..4.01 rows=201 width=17) (actual time=0.073..0.074 rows=201 loops=1)
        Buckets: 1024  Batches: 1  Memory Usage: 19kB
        ->  Seq Scan on "PayloadType" pt  (cost=0.00..4.01 rows=201 width=17) (actual time=0.008..0.036 rows=201 loops=1)
Planning Time: 0.645 ms
Execution Time: 3.299 ms
```

Only majorIOV

Investigating Query Plans - II

```
->  Limit  (cost=0.56..4.53 rows=1 width=61) (actual time=1.815..1.815 rows=1 loops=201)
      ->  Index Only Scan using combo_covering_idx on "PayloadIOV" pi
                (cost=0.56..3484.55 rows=876 width=61) (actual time=1.815..1.815 rows=1 loops=201)
          Index Cond: (payload_list_id = pl.id)
          Filter: ((major_iov < 100000000) OR ((major_iov = 100000000) AND (minor_iov <= 100000000)))
          Rows Removed by Filter: 24669
          Heap Fetches: 0
```

Index Condition & Filter

```
->  Limit  (cost=0.56..0.82 rows=1 width=61) (actual time=0.014..0.014 rows=1 loops=201)
      ->  Index Only Scan using combo_covering_idx on "PayloadIOV" pi
                (cost=0.56..232.55 rows=876 width=61) (actual time=0.013..0.013 rows=1 loops=201)
          Index Cond: ((payload_list_id = pl.id) AND (major_iov < 100000000))
          Heap Fetches: 0
```

Index Condition Only

History of the HSF CDB: road to ‘HSF product’

- sPHENIX needed CDB. Belle II’s solution lacked scalability
- HSF white paper suggested new DB schema w/ good scalability & payload agnostic
- Started to work on a reference implementation according to guidelines of that paper
  - In cooperation with HSF conditions data activity
  - Collect use cases & define minimal API
- Presented implementation and performance results at CHEP
  - Garnered attention and interest from HEP community
- Our implementation has been adopted for production use by sPHENIX
- Drove forward HSF integration, published source code, put it under Apache 2.0 license
  - Now listed as official ‘HSF product’ https://hepsoftwarefoundation.org/projects.html

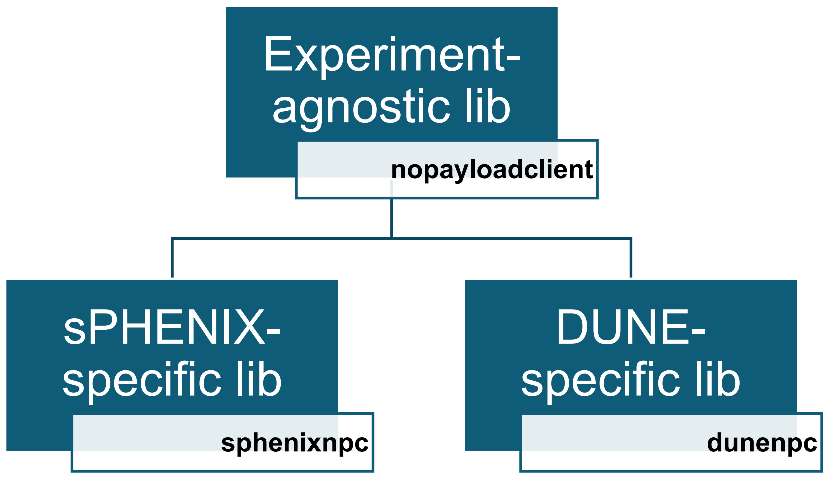
Features & Functionality

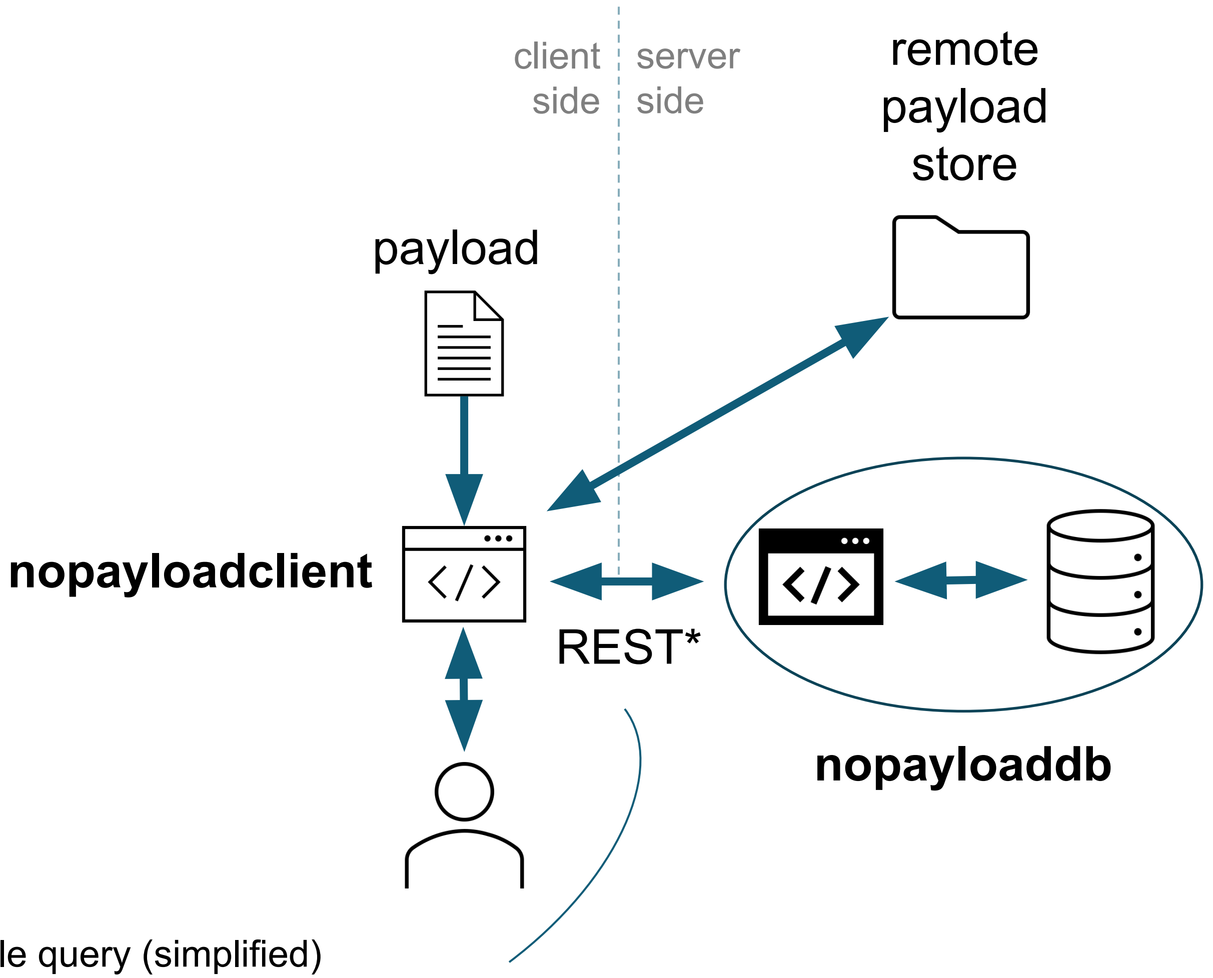
- Payload agnostic by design, loose server-client coupling (REST Interface)
- Proven scalability O(10M) payloads
- Easy deployment, configuration & horizontal scaling
- Standalone CLI & easy-to-integrate c++ client library
- Based completely on open source software:
    - Postgres, Django python API, c++ client library
    - Deployed on kubernetes and / or OKD/OpenShift, config via helm
- Integrated support of the common tag workflows
- Various caching options

HSF CDB Client

nopayloadclient:

- Client-side stand-alone C++ tool
- Communicates with **nopayloaddb** (server)
- Local caching
- Handling of payloads





*Example query (simplified)

```
curl http://<host>/api/payloadiovs/?gtName=test_gt&iovNum=42
-> {type_1: url_1, type_2: url_2, …}
```