Time series for European Space Agency Solar Orbiter Archive with TimescaleDB

Hector Perez, European Space Astronomy Centre, Madrid, Spain
David Kohn, Timescale, NYC, US

PGCONF.US 2018, 04/18/2018

ESA UNCLASSIFIED - For Official Use

Summary

- The ESA Solar Orbiter archive will provide interactive time series
- Feasibility study of the time series data in PostgreSQL comparing
    - PostgreSQL 10 non-partitioned
    - PostgreSQL 10 partitioned
    - PostgreSQL 10 + TimescaleDB partitioned
- **Conclusion**: The benchmarks obtained with the latter are good for the Solar Orbiter use case and TimescaleDB eases the maintenance of partitions

Through the following slides we will explain the required context and the process followed, providing an insight on TimescaleDB partitioning.

ESAC Science Data Centre (ESDC)
The Digital Library of the Universe

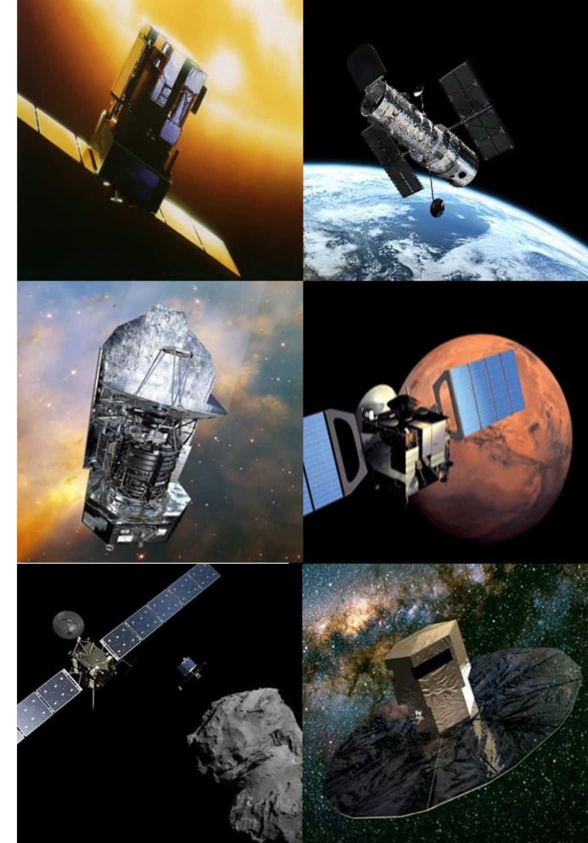
At ESA's European Space Astronomy Centre near Madrid

Science Archives from ~20 space missions:

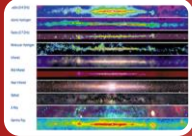
- Astronomy, Planetary, Heliophysics
- From all phases (development, operations, post-ops, legacy)
- http://archives.esac.esa.int/

Different Users:

- Scientific Community (public access)
- Instrument teams and observers (controlled access)
- Science Operations Team (privileged access)

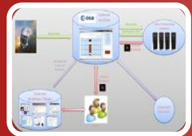


Long Term Strategy for Science Archives at ESAC



Enable maximum ***scientific exploitation*** of data sets

Enable efficient ***long-term preservation*** of data, software and knowledge, using modern technology



Enable cost-effective archive production by ***integration in, and across, projects***

ESAC SCIENCE DATA CENTRE

Astronomy Science Archives

esasky
exosat
gaia
herschel
hubble space telescope
iso
lisa pathfinder
planck
xmm-newton

Heliophysics Science Archives

cluster
double star
ISS-SolACES*
proba-2
soho
ulysses

The Planetary Science Archive

cassini huygens
exomars
giotto
mars express
rosetta
smart-1
venus express

Future Archives

bepicolombo
euclid
juice
solar orbiter

ESA UNCLASSIFIED - For Official Use

ESA Space Science Archives - Volume (PB)

1.60
1.40
1.20
1.00
0.80
0.60
0.40
0.20
0.00

2005
2010
2011
2012
2013
2014
2015
2016
2019

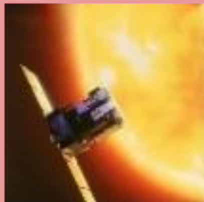
Euclid will add 45PB by 2025

- Gaia - 2013
- Herschel - 2009
- Planck - 2009
- Planetary (MEX - 2003, Rosetta - 2004, VEX - 2005, BepiC - 2018)
- Cluster - 2000
- XMM-Newton - 1999
- Soho - 1995
- Hubble - 1990

ESA Heliophysics Science Archives

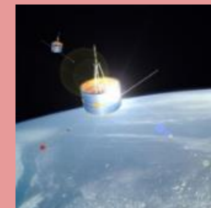
Operational

Cluster


Soho

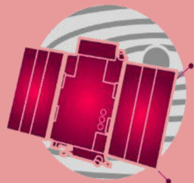
Ulysses

ISS-Solaces

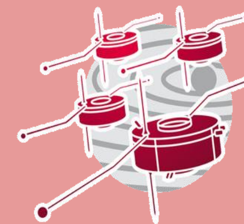

Double Star

Proba-2

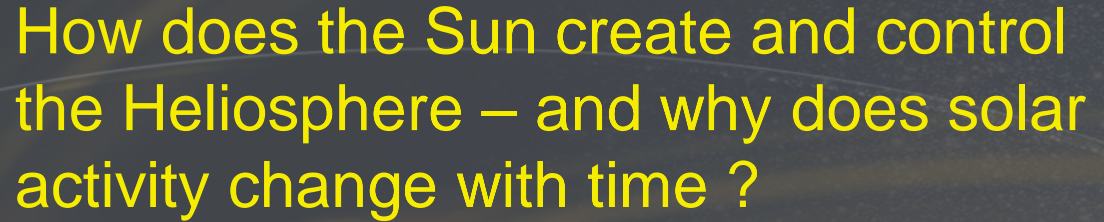
In-development


Proba-2
Second of ESA's PRoject for OnBoard Autonomy satellites

Solar Orbiter
Exploring the Sun-Heliosphere connection


Cluster 2.0
Sun-Earth Environment

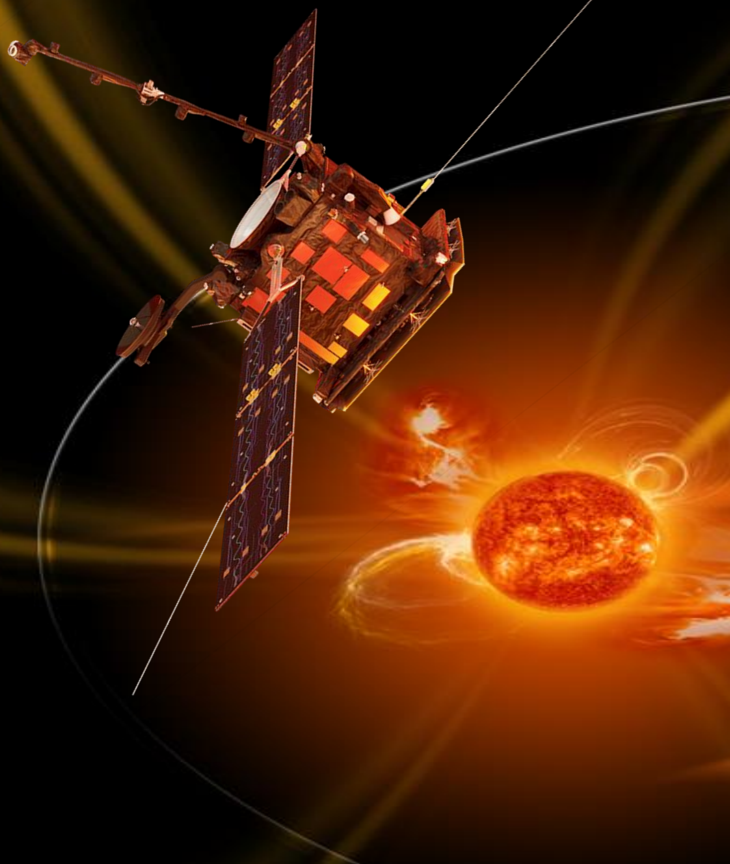
Solar Orbiter - Science objectives



1. What drives the solar wind and where does the coronal magnetic field originate?
2. How do solar transients drive heliospheric variability?
3. How do solar eruptions produce energetic particle radiation that fills the heliosphere?
4. How does the solar dynamo work and drive connections between the Sun and the heliosphere?

Mission overview: Müller et al., *Solar Physics* **285** (2013)

The Solar Orbiter mission summary



Launch: 2020

Cruise Phase: 2.3 years

Nominal Mission: 3.5 years

Extended Mission: 2.5 years

Orbit: 0.28–0.91 AU (P=150-180 days)

→ Unprecedented sun pole views

→ Closest mission to the sun

→ Linking in-situ and remote sensing observations

Solar Orbiter ARchive (SOAR)

Web application

- Store Solar Orbiter data
- Access to metadata
- Download files
- VO Interoperability
- Command line
- **Time series visualisation**
  - Highcharts
  - GWT

Solar Orbiter Archive

SOAR 0.4

RESULTS #1

low_latency (81)

Item Id	Level	Descriptor	SC Begin Tim	SC End Time	Instrument
solo_LL01_spice-w-sit_0481294961	LL01	SPICE-W-SIT	0481294961	0481381361	SPICE
solo_LL01_spice-w-sit_0481294974	LL01	SPICE-W-SIT	0481294974	0481381374	SPICE
solo_LL01_spice-n-sit_0481295089	LL01	SPICE-N-SIT	0481295089	0481381489	SPICE
solo_LL01_spice-n-sit_0481295332	LL01	SPICE-N-SIT	0481295332	0481381732	SPICE
solo_LL01_spice-n-sit_0481295390	LL01	SPICE-N-SIT	0481295390	0481381790	SPICE
solo_LL01_spice-n-exp_0481296410	LL01	SPICE-N-EXP	0481296410	0481382810	SPICE
solo_LL01_epd-het-north-rates_0699494400-0699580799	LL01	EPD-HET-NORTH-RATES	0699494400	0699580799	EPD
solo_LL01_epd-ept-north-rates_0699494400-0699580799	LL01	EPD-EPT-NORTH-RATES	0699494400	0699580799	EPD
solo_LL01_epd-het-asun-rates_0699494400-0699580799	LL01	EPD-HET-ASUN-RATES	0699494400	0699580799	EPD
solo_LL01_epd-sis-B-rates_0699494400-0699580799	LL01	EPD-SIS-B-RATES	0699494400	0699580799	EPD
solo_LL01_epd-ept-south-rates_0699494400-0699580799	LL01	EPD-EPT-SOUTH-RATES	0699494400	0699580799	EPD
solo_LL01_epd-sis-A-rates_0699494400-0699580799	LL01	EPD-SIS-A-RATES	0699494400	0699580799	EPD
solo_LL01_epd-step-rates_0699494400-0699580799	LL01	EPD-STEP-RATES	0699494400	0699580799	EPD
solo_LL01_epd-het-sun-rates_0699494400-0699580799	LL01	EPD-HET-SUN-RATES	0699494400	0699580799	EPD
solo_LL01_epd-ept-sun-rates_0699494400-0699580799	LL01	EPD-EPT-SUN-RATES	0699494400	0699580799	EPD
solo_LL01_epd-ept-asun-rates_0699494400-0699580799	LL01	EPD-EPT-ASUN-RATES	0699494400	0699580799	EPD
solo_LL01_epd-het-south-rates_0699494400-0699580799	LL01	EPD-HET-SOUTH-RATES	0699494400	0699580799	EPD

1 of 1 Page size: 100 Displaying 1-81 of 81

SOLO_LL01_EPD-HET-NORTH-RATES_0699494400-

Science CDF Details | Variable Details | Repository Files

DATA

- ELE_FLUX (Electron Flux)
- ELE_FLUX_SIGMA (Electron Flux Sigma)
- PROT_FLUX (Proton Flux)
- PROT_FLUX_SIGMA (Proton Flux Sigma)
- HE3_FLUX (Helium-3 Flux)
- HE3_FLUX_SIGMA (Helium-3 Flux Sigma)
- HE4_FLUX (Helium-4 Flux)
- HE4_FLUX_SIGMA (Helium-4 Flux Sigma)
- CNO_FLUX (Combined CNO Flux)
- CNO_FLUX_SIGMA (Combined CNO Flux Sigma)
- FE_FLUX (Iron Flux)
- FE_FLUX_SIGMA (Iron Flux Sigma)
- ELE_HR_FLUX (Electron High Resolution Flux)
- ELE_HR_FLUX_SIGMA (Electron HR Flux Sigma)
- PROT_HR_FLUX (Proton High Resolution Flux)
- PROT_HR_FLUX_SIGMA (Proton HR Flux Sigma)

METADATA

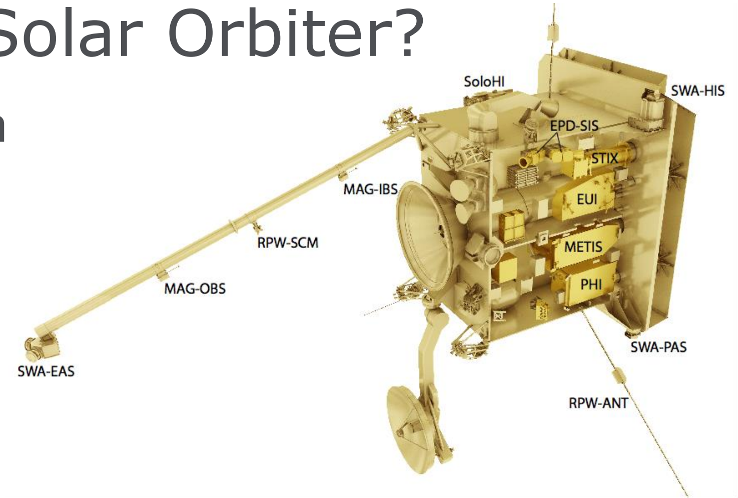
SUPPORT_DATA

COPYRIGHT © EUROPEAN SPACE AGENCY. ALL RIGHTS RESERVED.

Solar Orbiter data

What kind of data we get from Solar Orbiter?

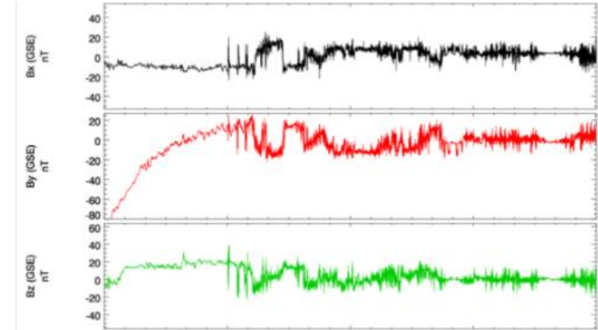
- Low latency (LL01, LL02) + science data
- Remote sensing images + in-situ data
- Data stored in FITS and CDF files
- In-situ data → mainly in CDF files
- Early stage → lack of data examples



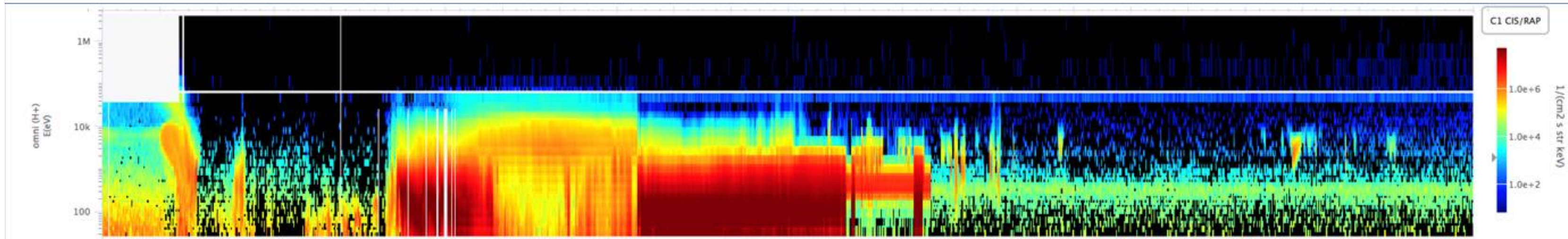
Solar Orbiter time series data

CDF Data variables

- All are time varying
- Other dependencies may exist
- 0 extra dependencies → array/timelines



- 1, 2 or 3 extra dependencies → matrix/spectrograms



LL time series data estimations (10y)

Dataset	Files size (GB)	# Rows (M)	Table Size (GB)
SWA_EAS	6.2	3	7.5 *
SWA_PAS	3.5	78	195 *
SWA_HIS	5.5	10	25 *
MAG	1.7	39	97.5 *
EPD_STEP_RATES	9.8	315	787.5 *
EPD_EPT_RATES	18.9	63	157.5 *
EPD_SIS_RATES	2.1	0.2	0.5 *
EPD_HET_RATES	16.0	63	157.5 *
TOTAL	**63.73**	**-**	**1428 ***

* Extrapolation for 1 variable of 1 dataset, with 2DEP (15*4)

Time series storage/usage requirements

Time series data: RDBMs or NoSQL Time Series DBs?

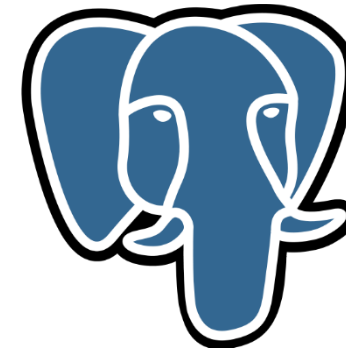
Time series requirement	RDBMS or TSDB
Structured data types	Enough with RDBMS
Need of powerful query language	Easy with (and used to) RDBMS
No need of high insert throughput	Enough with RDBMS
Volume of time series is medium	**Enough with RDBMS? Scalability?**

→ Benchmarks to evaluate feasibility with RDBMS

→ What RDBMS?

Database systems at ESDC

Generally used in the archives: **PostgreSQL**: 9.0 to 10.0



- Opensource
- Big community
- Extensions for spherical queries (pg_sphere, q3c, healpix, postgis)
- Professional companies support

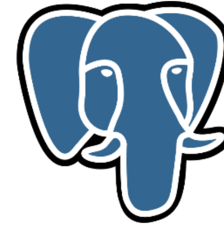
Exceptions:

- ISO: **Sybase**
- Euclid DPS: **Oracle** -> **PostgreSQL?**
- ESASky, PSA (some functionalities): **ElasticSearch**

Feasibility study

Comparison of technologies

- PostgreSQL 10 non-partitioned
- PostgreSQL 10 partitioned
- PostgreSQL 10 + **TimescaleDB** partitioning



Why TimescaleDB?

- PostgreSQL plugin
- Time series oriented
- Easy-to-use partitioning
- Looks promising for our purpose

TimescaleDB:

Scalable time-series database, full SQL

Packaged as a PostgreSQL extension

Apache 2 Licensed

Scalable time-series database, full SQL

Packaged as a PostgreSQL extension

Easy to Use

- Supports full SQL
- Time-oriented features
- Easy to manage: looks like a regular table
- One DB for relational & time-series data

Scalable

- High write rates
- Time-oriented optimizations
- Fast complex queries
- 100s billions rows / node

Reliable

- Inherits 20+ years of PostgreSQL reliability
- Streaming replication, backups, HA clustering

How?

Time-series workloads are different.

OLTP

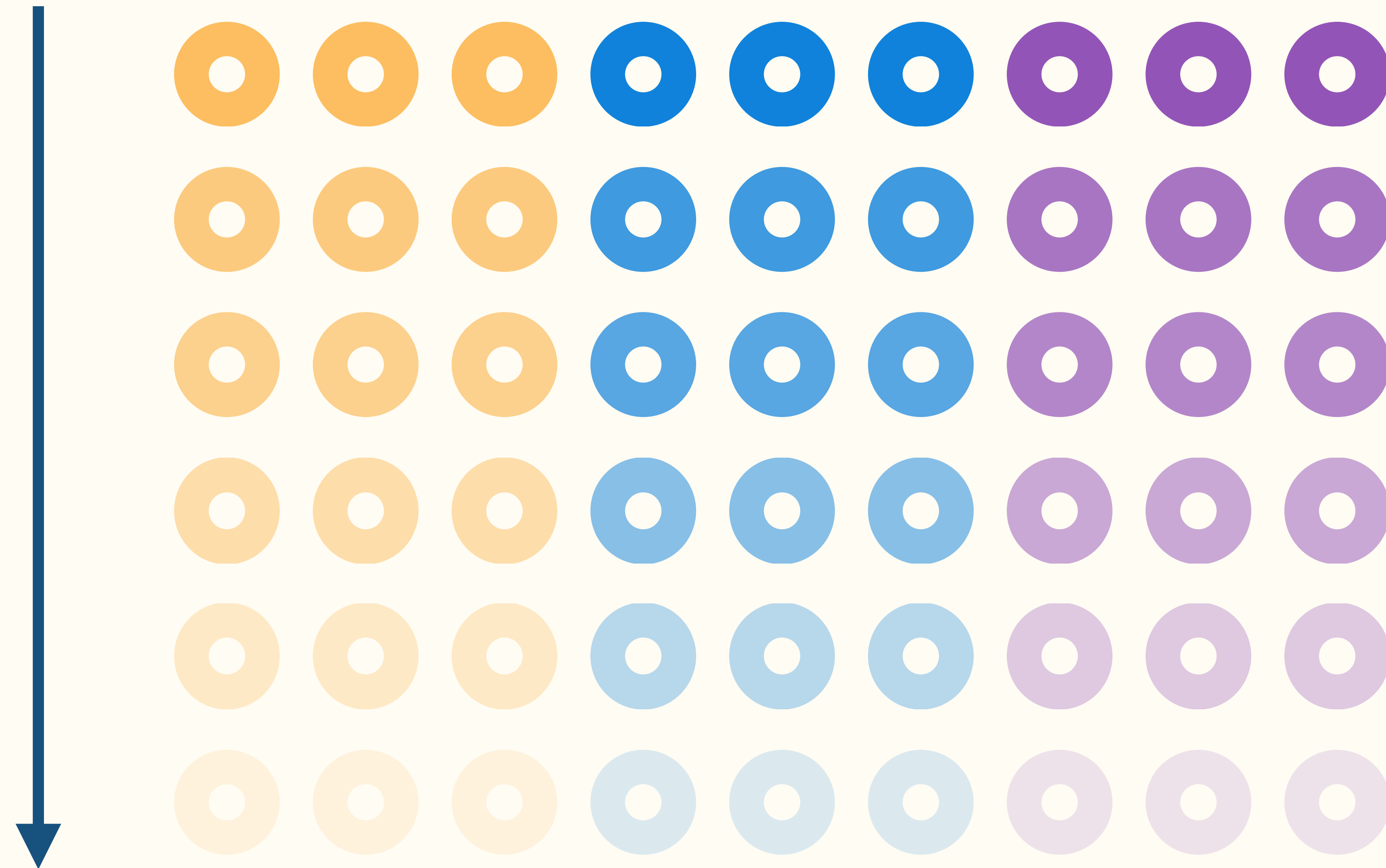
- 👎 Primarily UPDATEs
- 👎 Writes randomly distributed
- 👎 Transactions to multiple primary keys

Time-series

- 👍 Primarily INSERTs
- 👍 Writes to recent time interval
- 👍 Writes associated with a timestamp and primary key

How it works

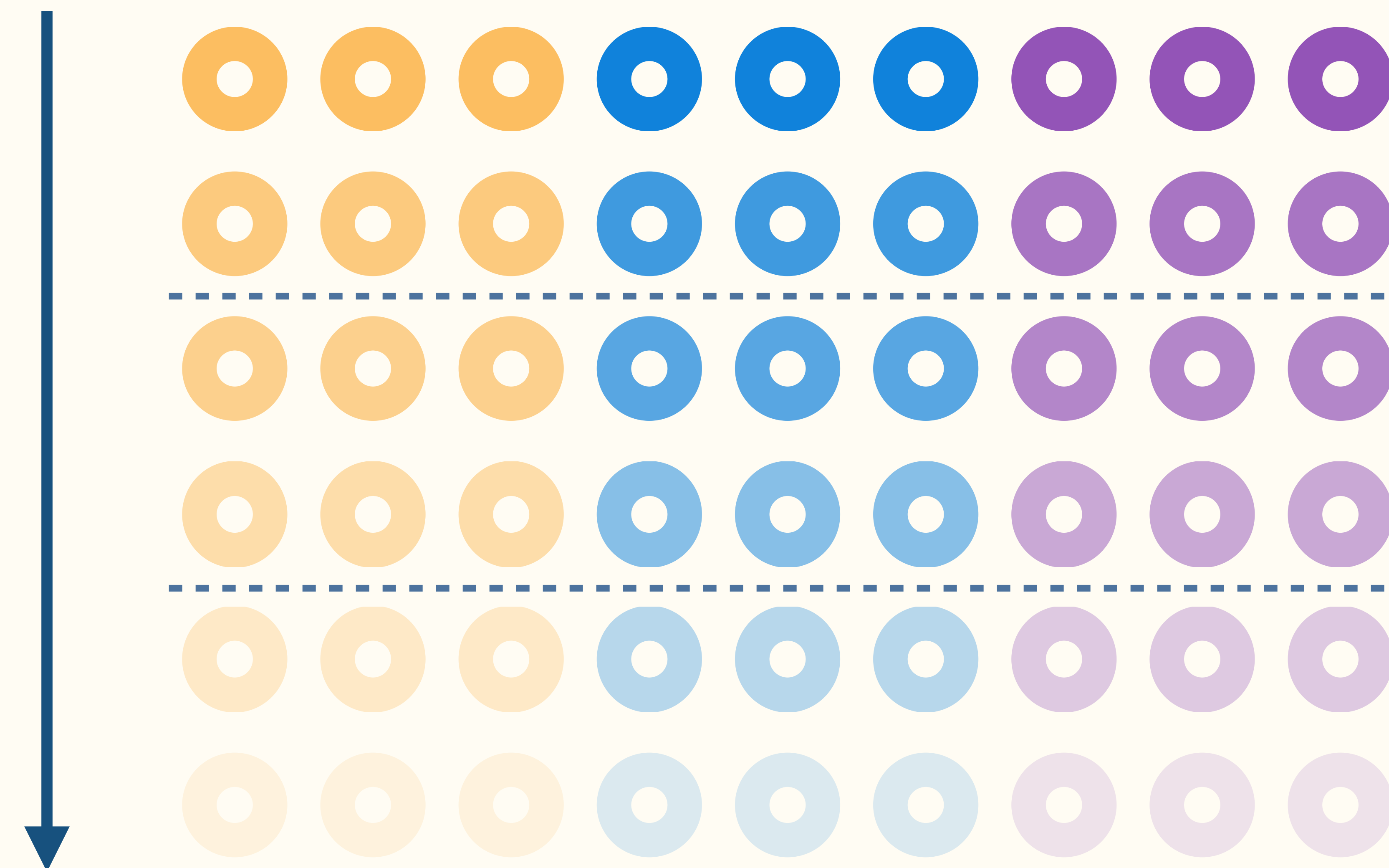
Time
(older)



Time-space partitioning

(for both scaling up & out)

Time
(older)



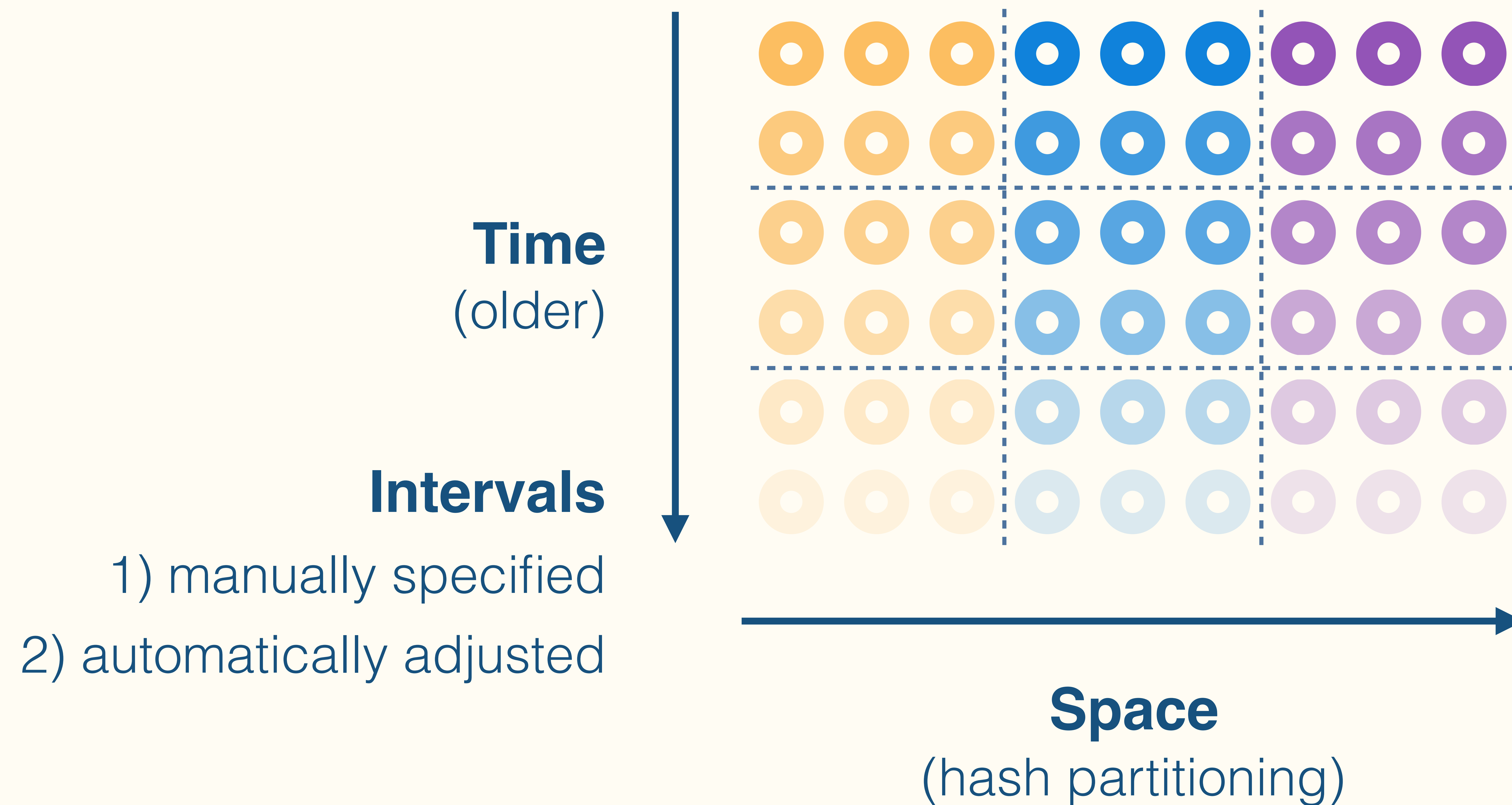
Intervals

1) manually specified

2) automatically adjusted

Time-space partitioning

(for both scaling up & out)



Time-space partitioning

(for both scaling up & out)

Time
(older)

Intervals

1) manually specified

2) automatically adjusted

Chunk (sub-table)

Space
(hash partitioning)

Time-space partitioning

(for both scaling up & out)

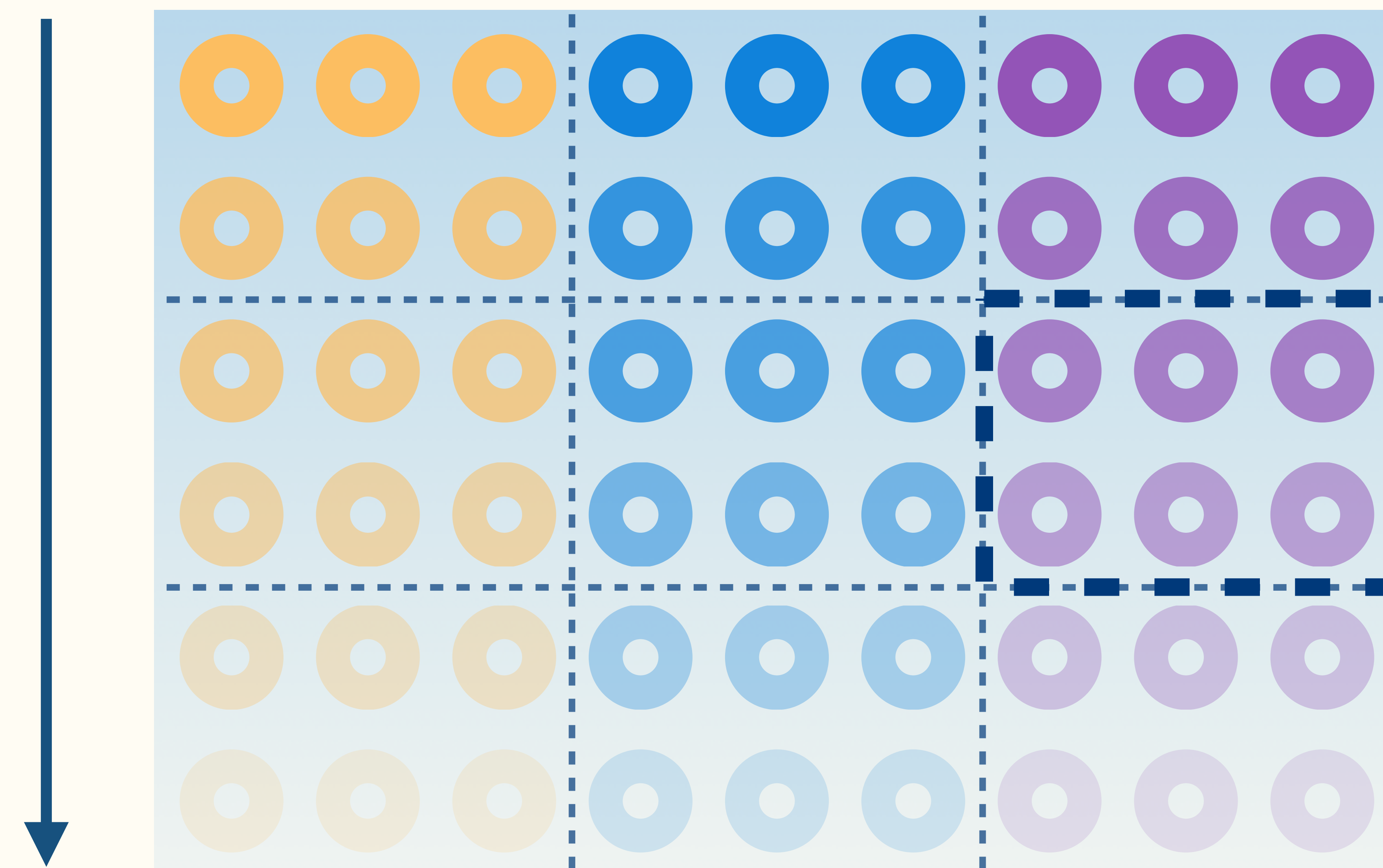
Hypertable

Time
(older)

Intervals

1) manually specified

2) automatically adjusted



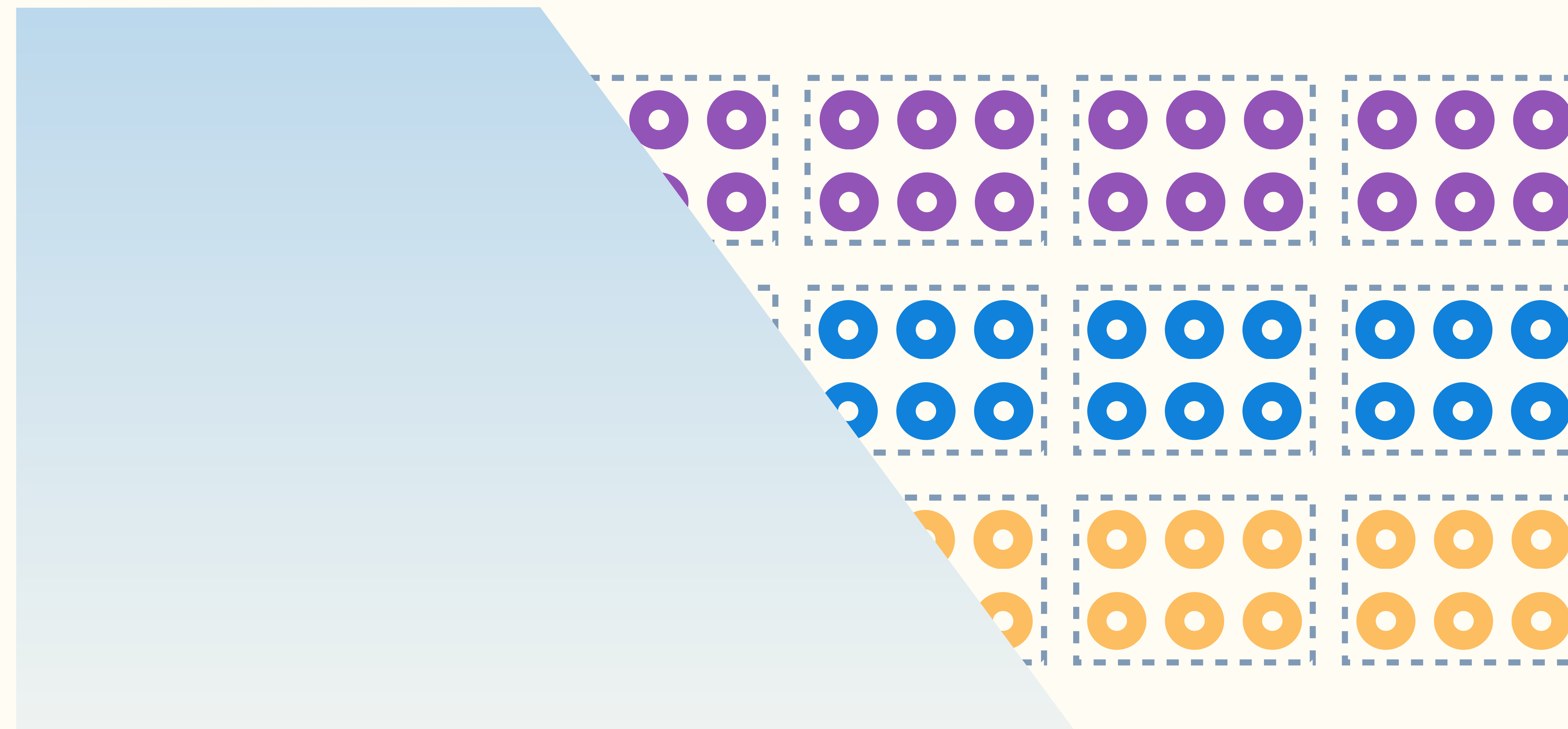
Chunk (sub-table)

Space
(hash partitioning)

The Hypertable Abstraction

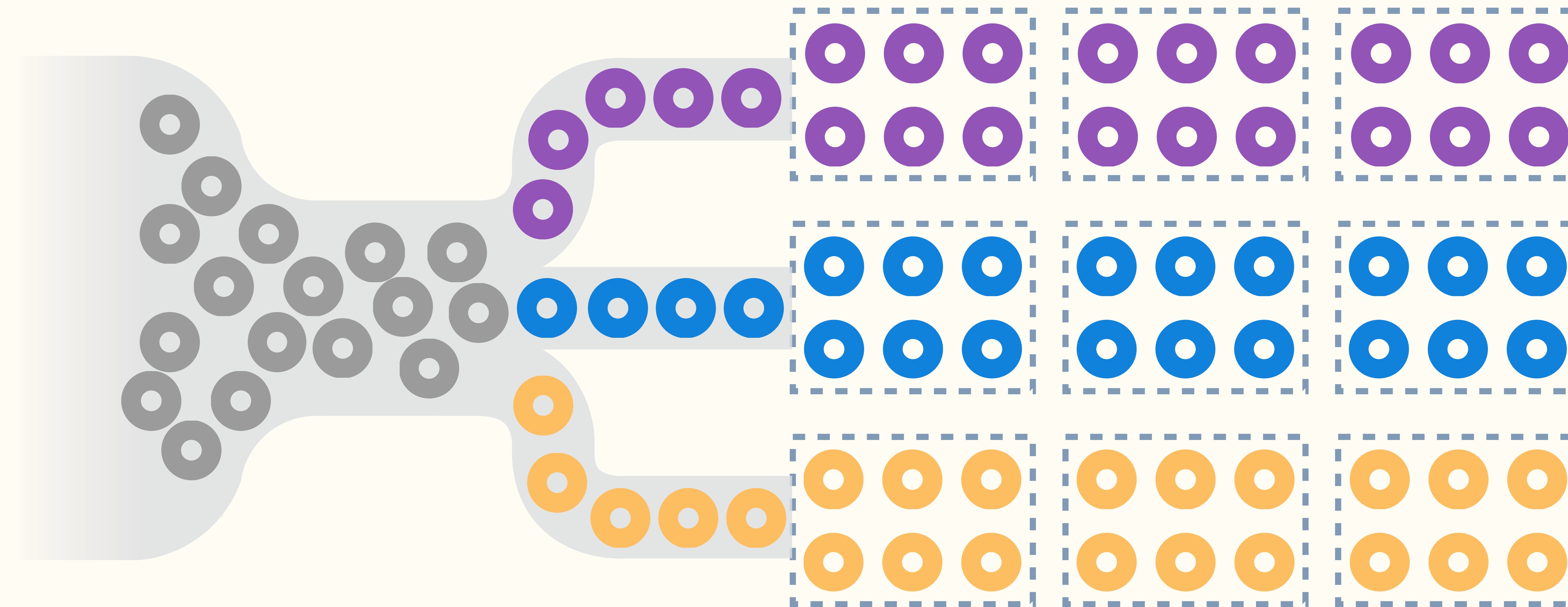
Hypertable

- Triggers
- Constraints
- Indexes
- UPSERTs
- Table mgmt

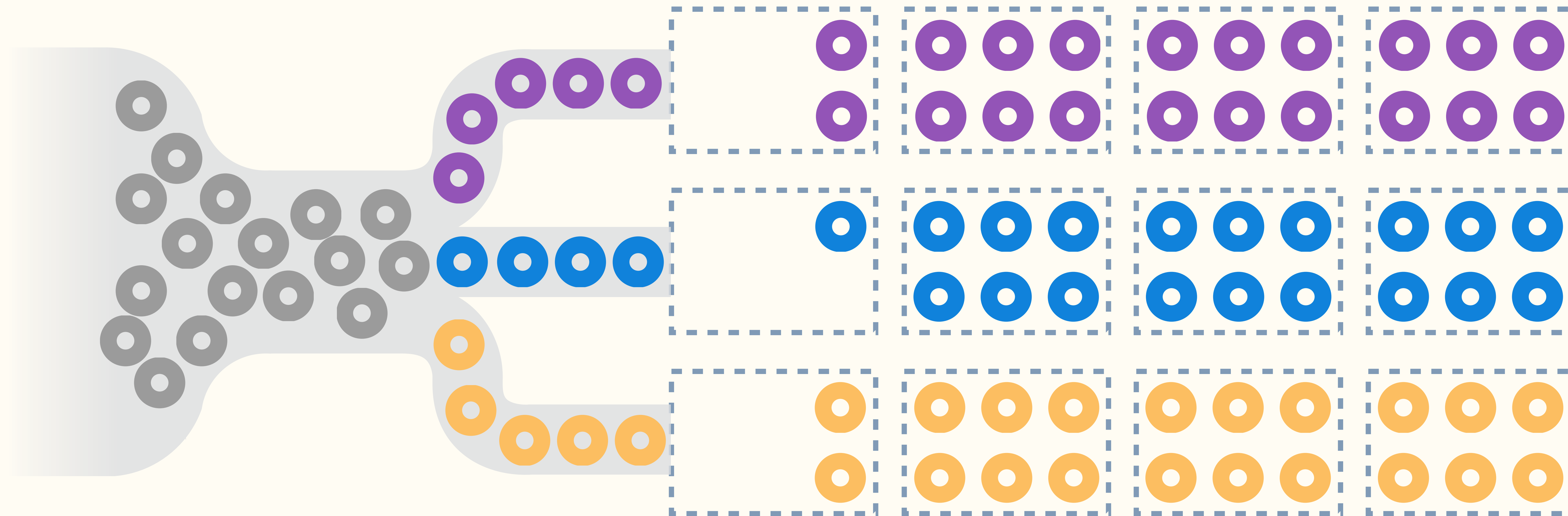


Chunks

Automatic Space-time Partitioning

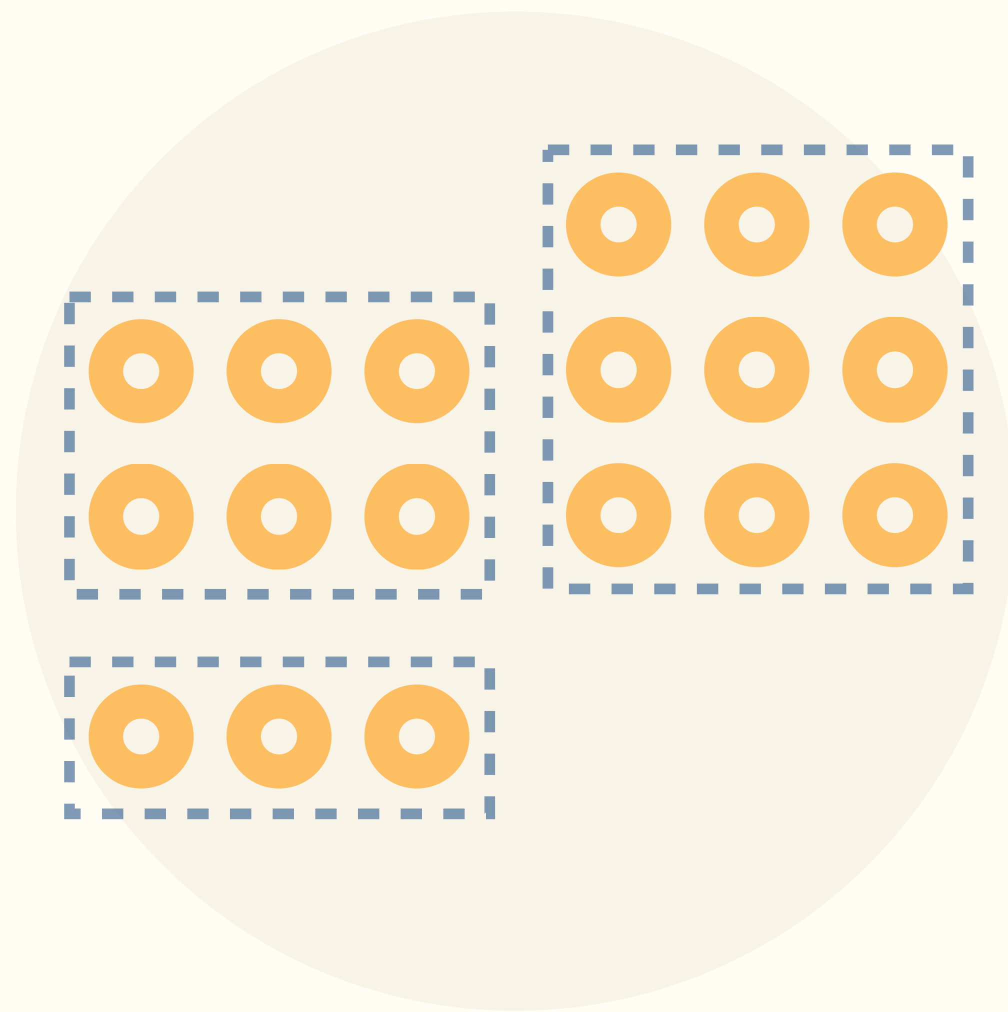
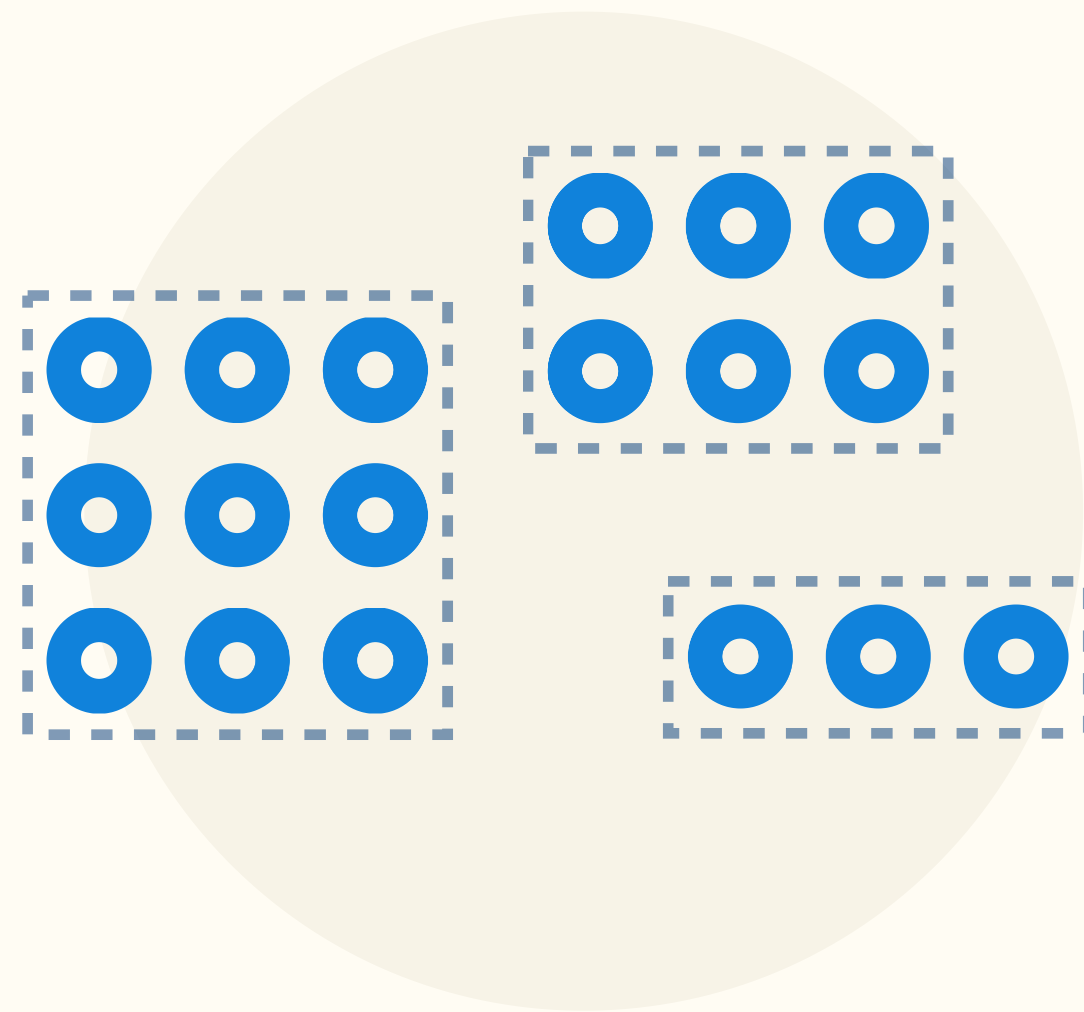
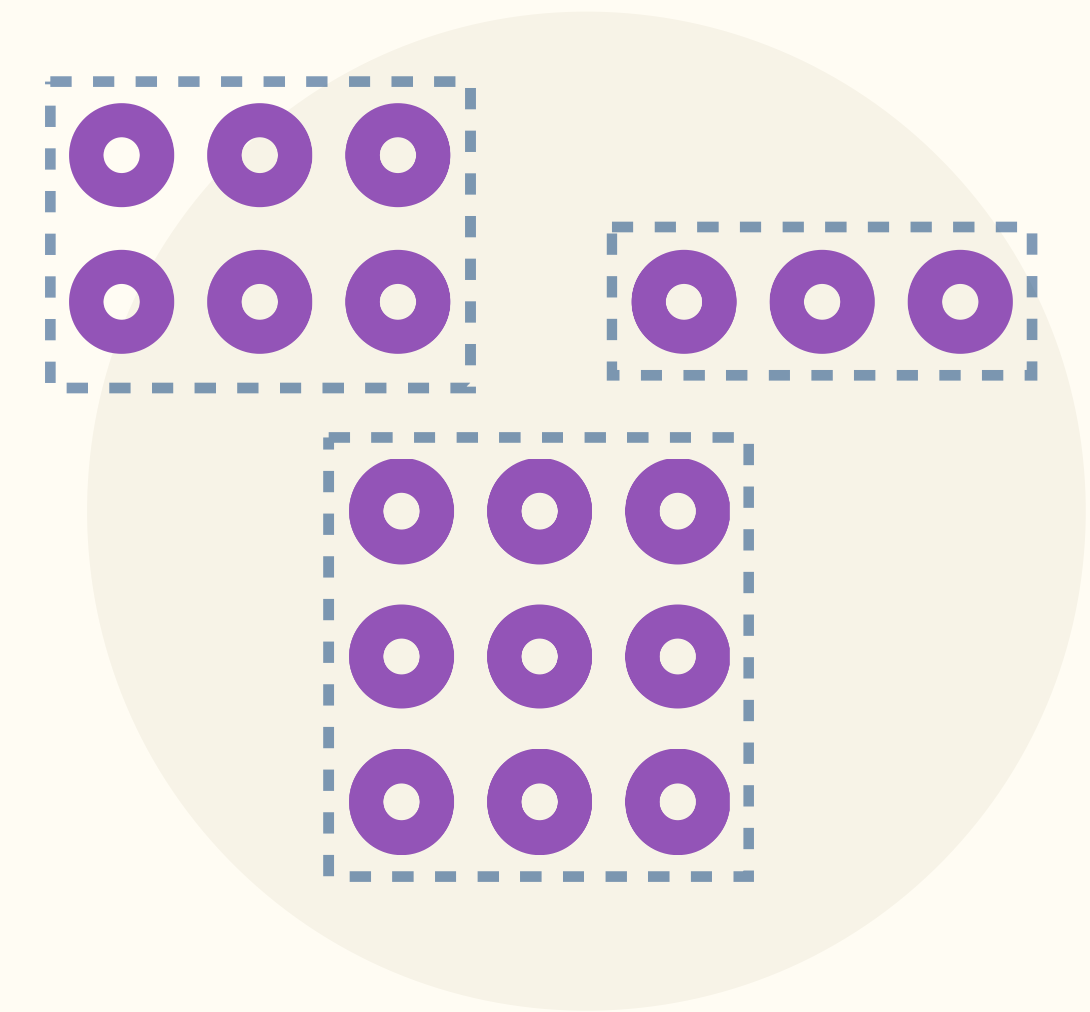


Automatic Space-time Partitioning



Chunking benefits

Chunks are **“right-sized”**

Recent (hot) chunks fit in memory

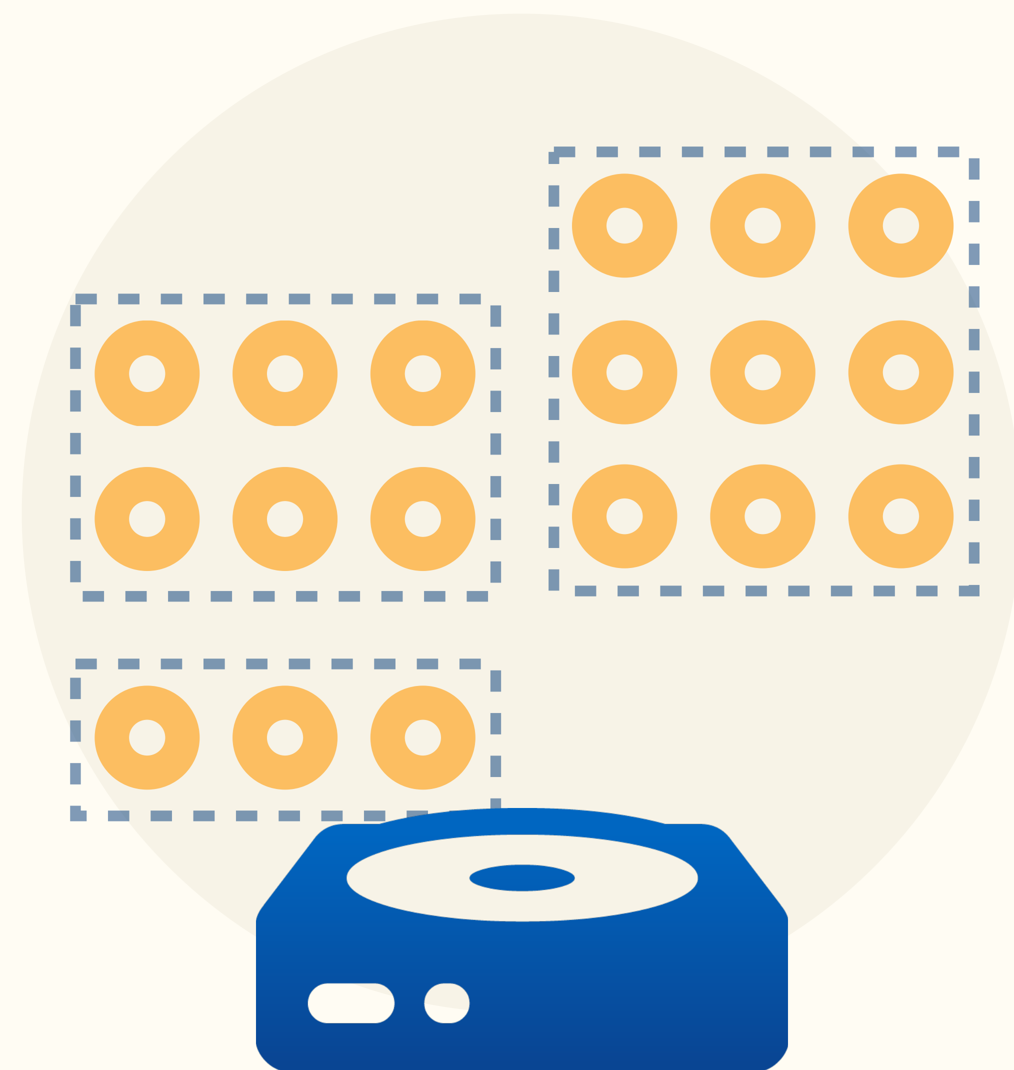
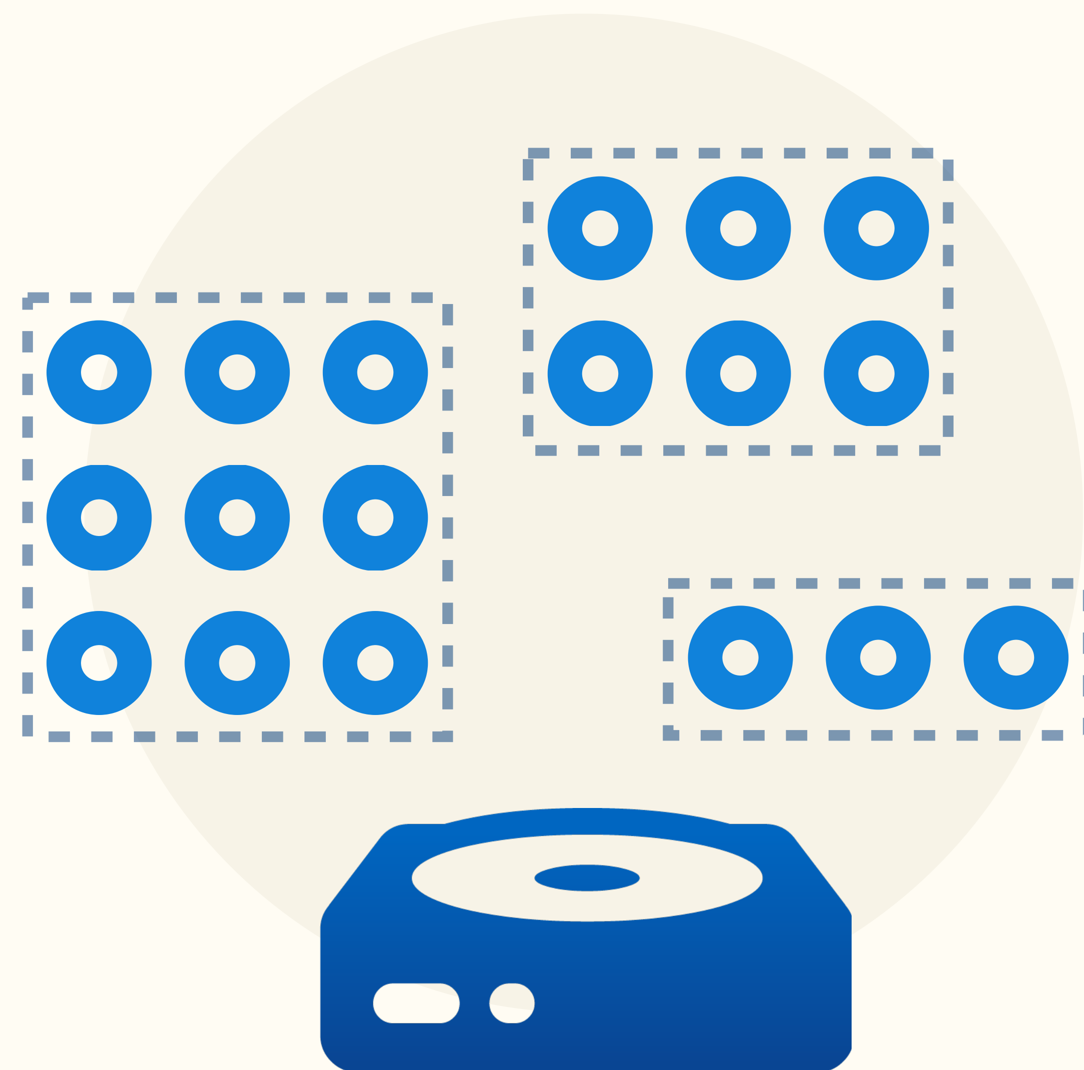
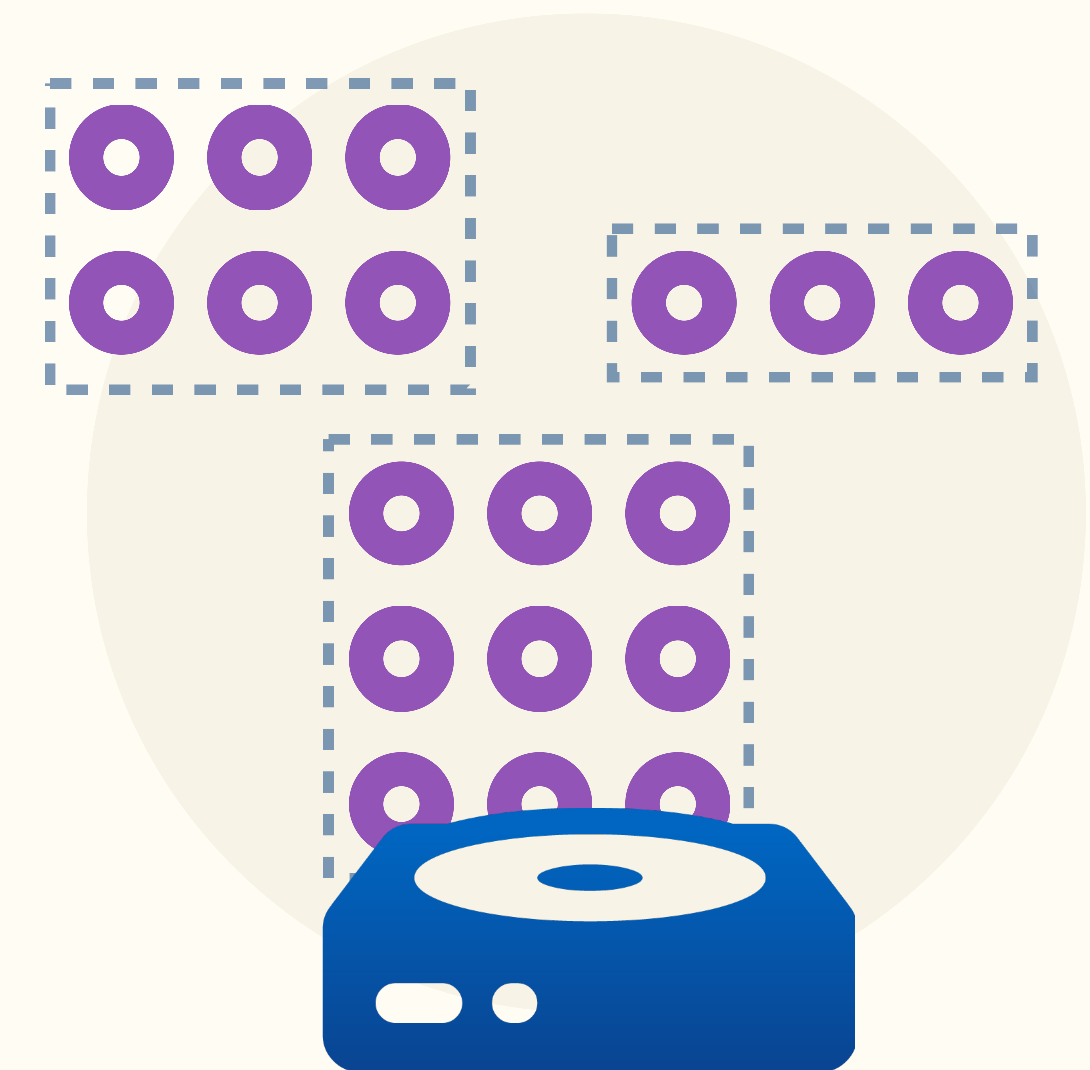
Single node: **Scaling up via adding disks**

How

- Chunks spread across many disks (elastically!)
  either RAIDed or via distinct tablespaces

Benefit

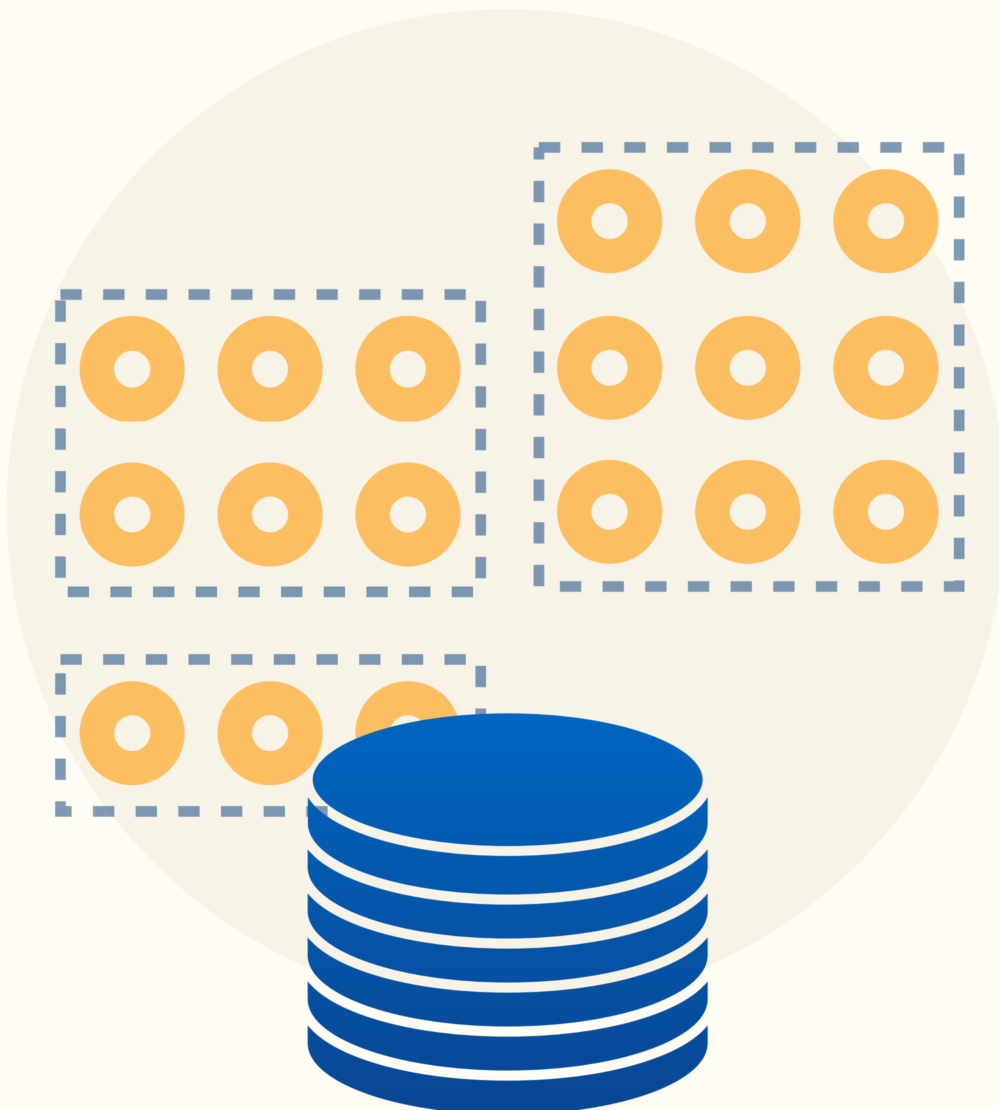
- Faster inserts
- Parallelized queries

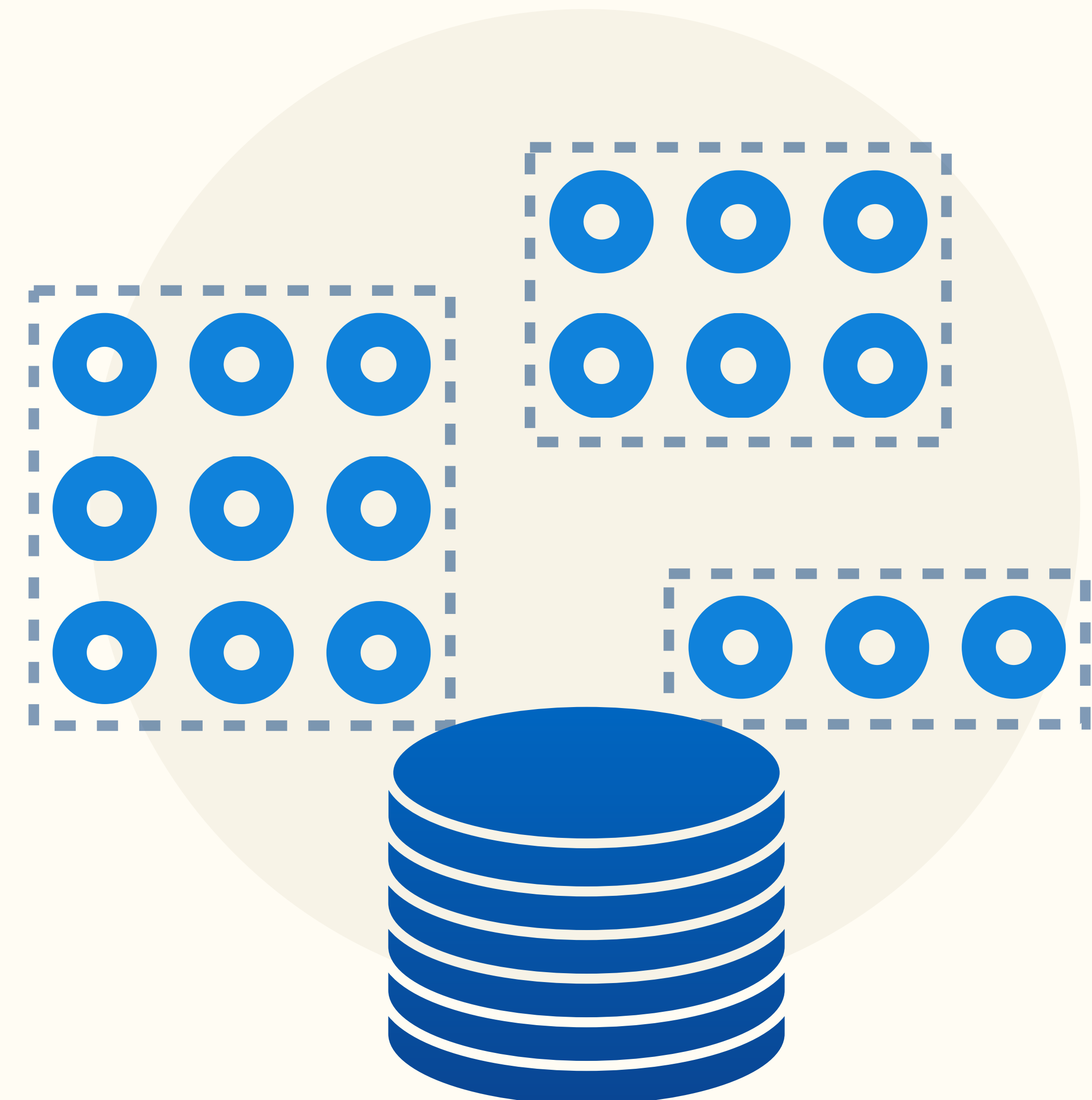




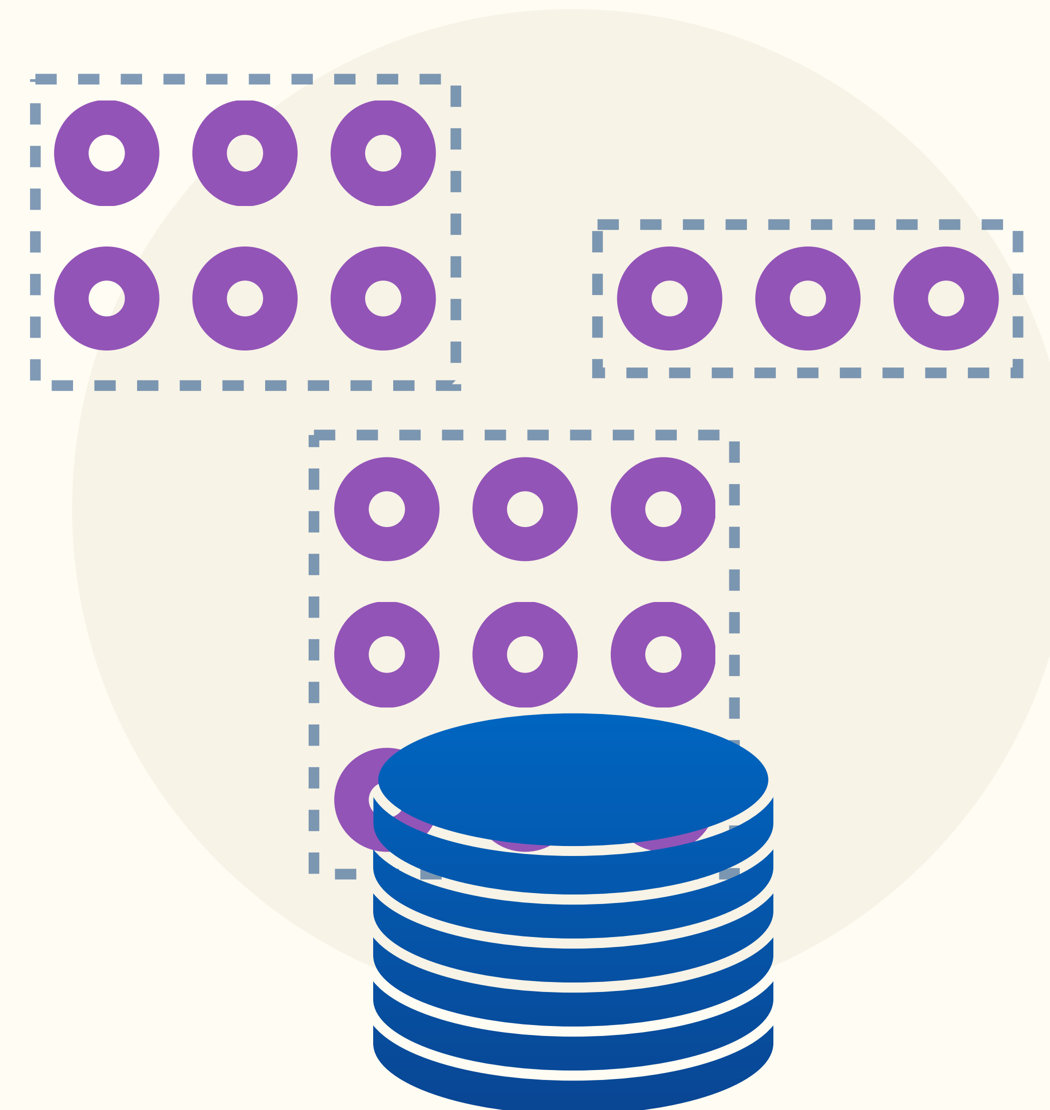
Under development

Multi-node: **Scaling out via existing mechanisms**

- Chunks spread across servers
- Insert/query to any server
- Distributed query optimizations
  (push-down LIMITs and aggregates, etc.)

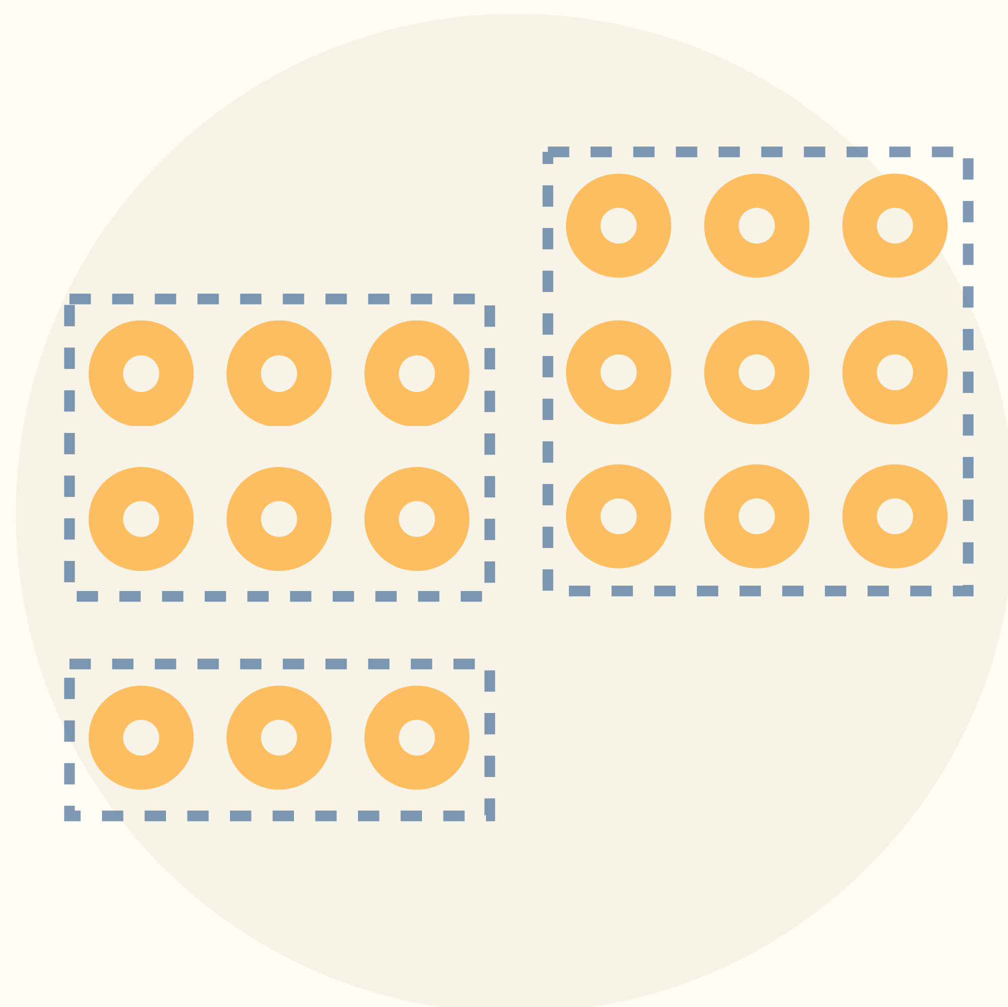


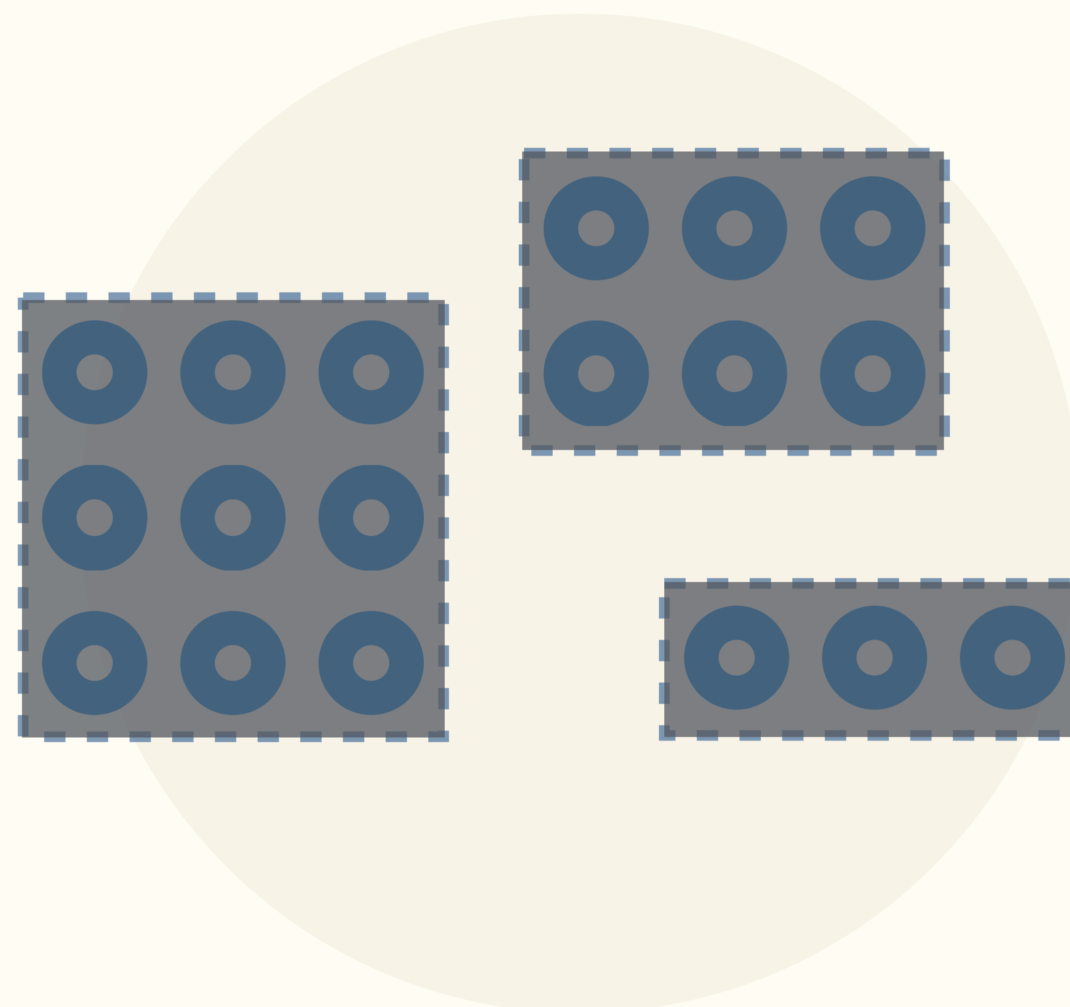


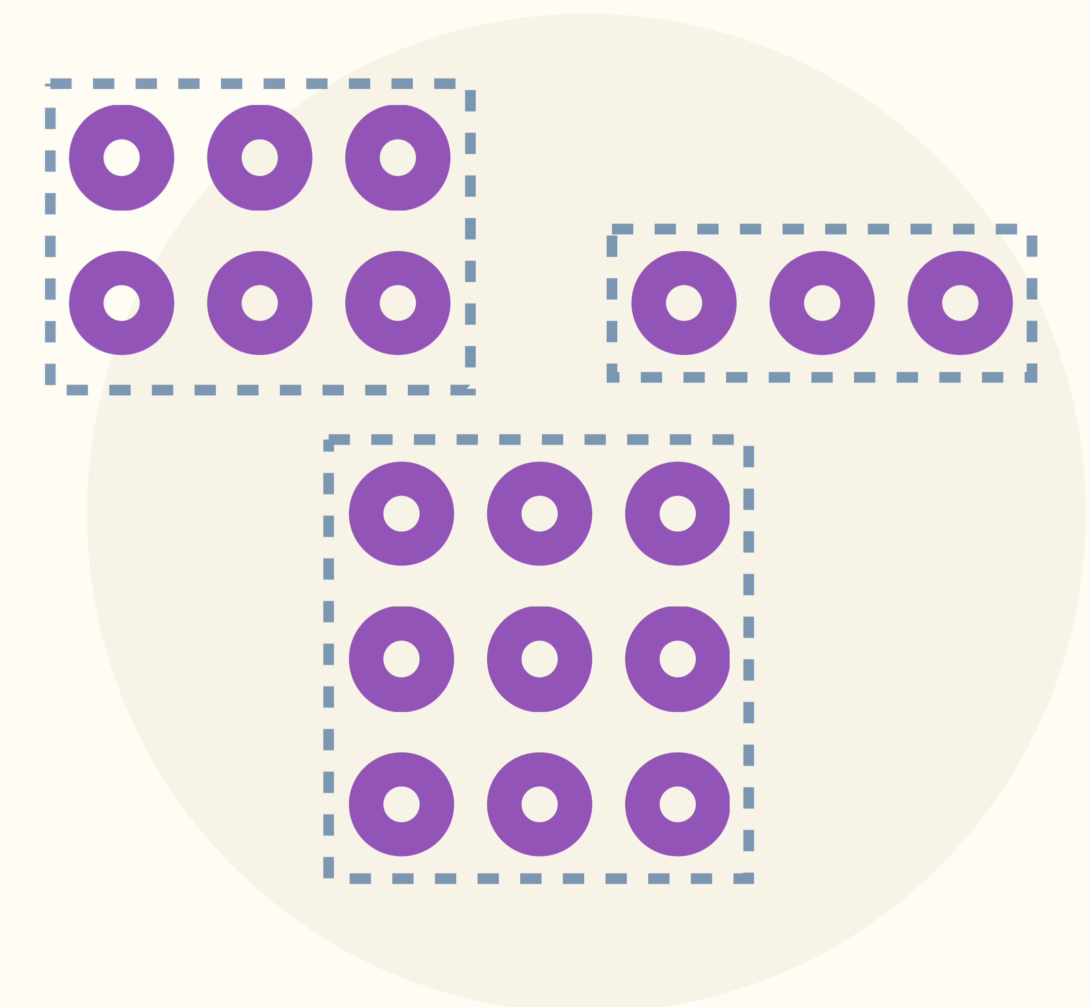


Avoid querying chunks via **constraint exclusion**

```
SELECT time, temp FROM data
   WHERE time > now() - interval ‘7 days
      AND device_id = ‘12345’
```

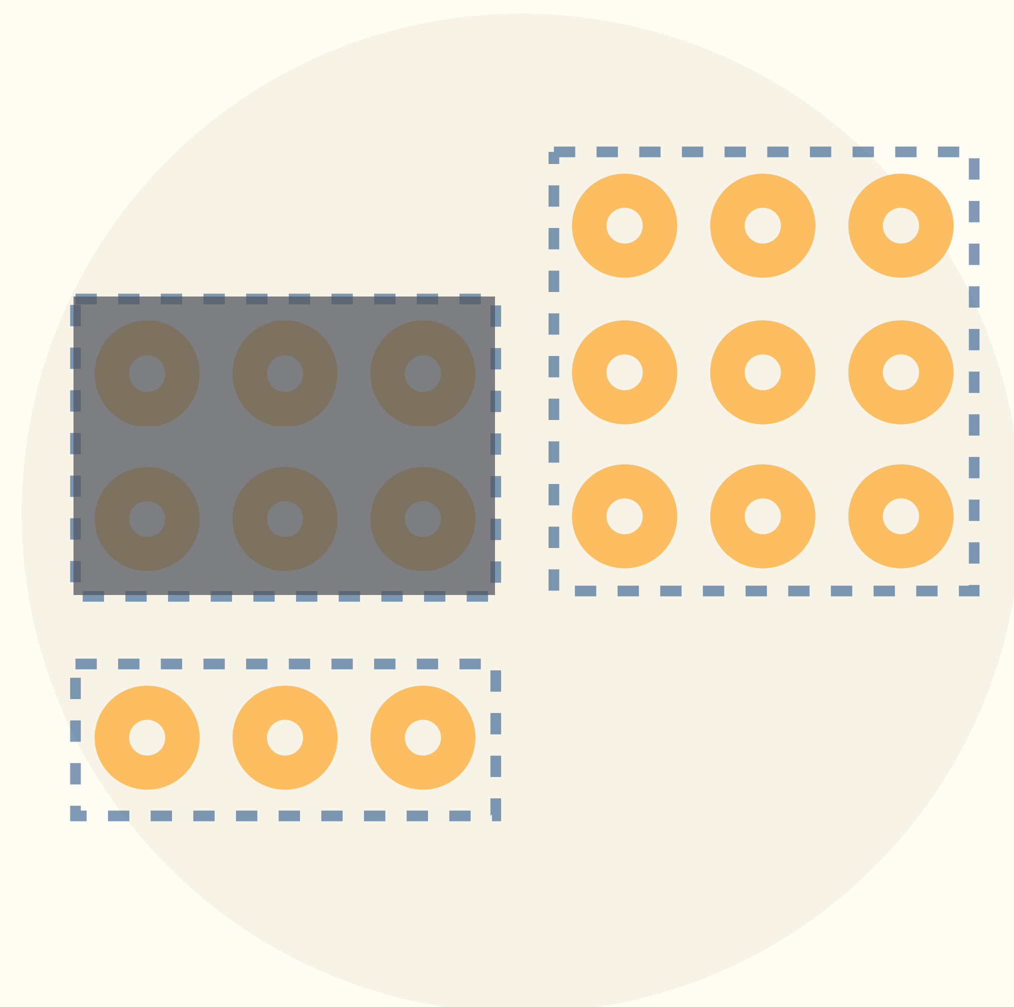


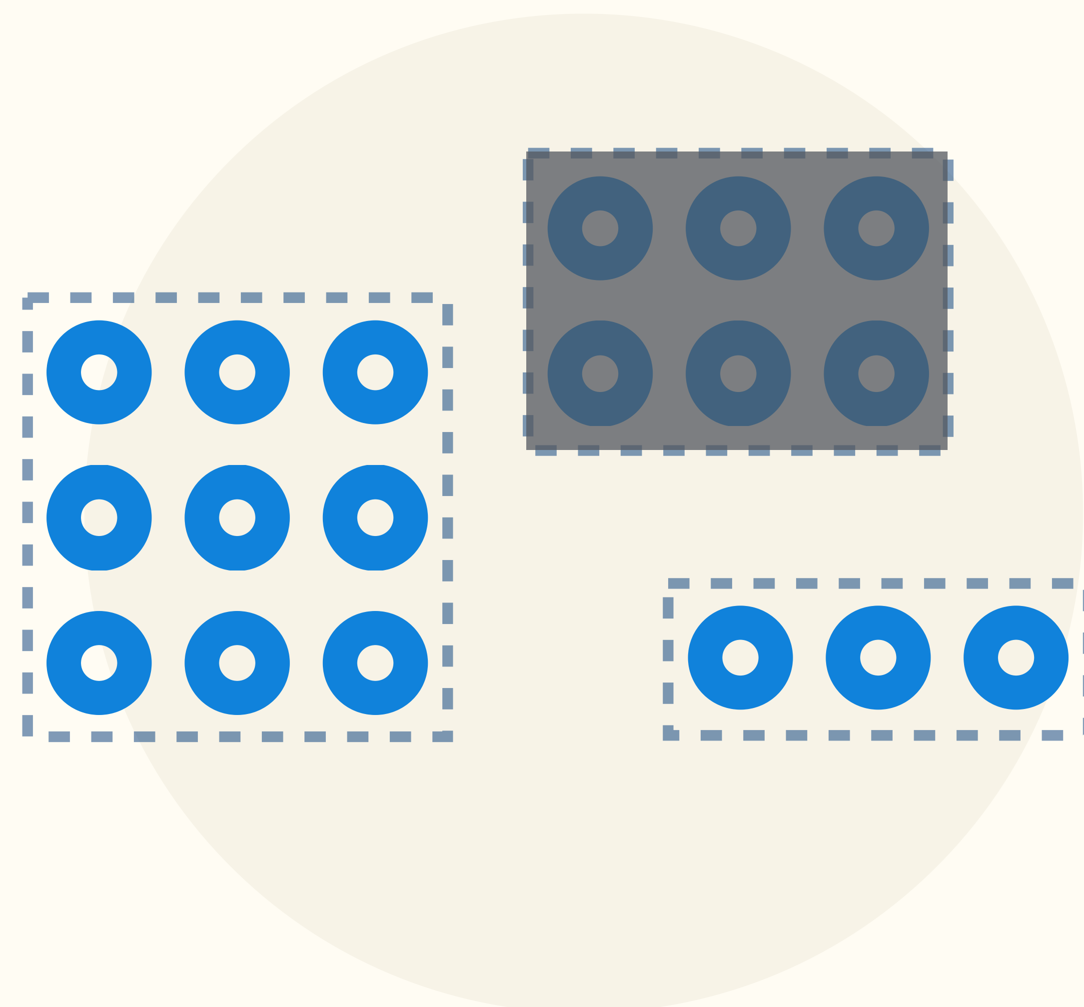


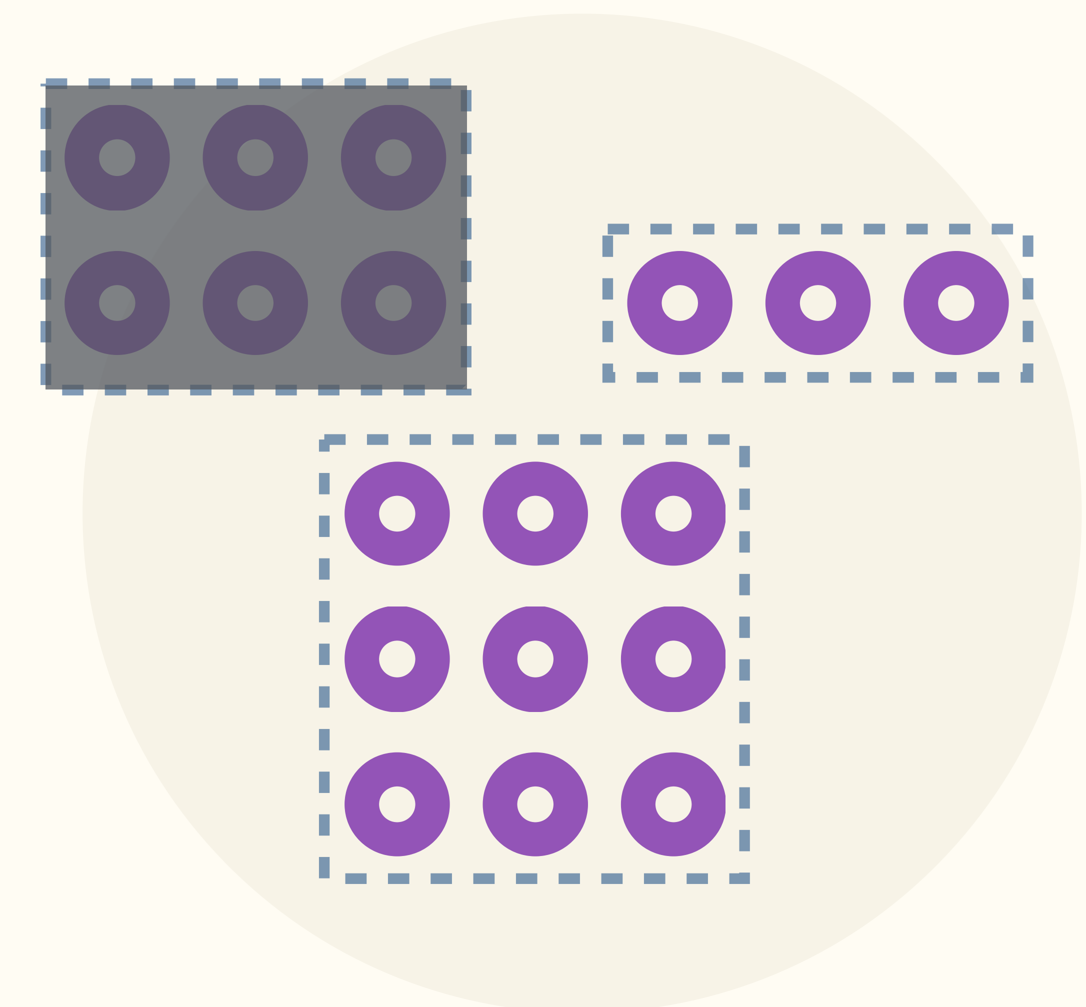


Avoid querying chunks via **constraint exclusion**

```
SELECT time, device_id, temp FROM data
   WHERE time > '2017-08-22 18:18:00+00'
```



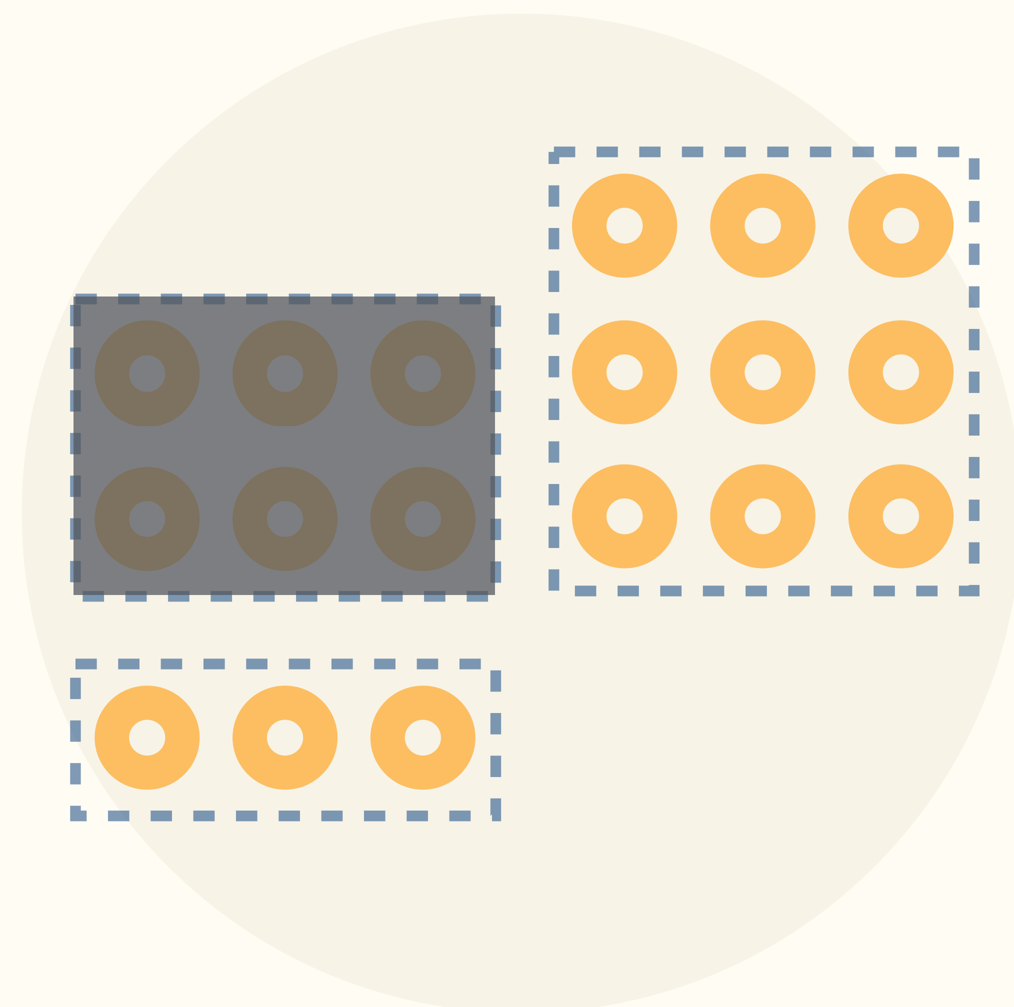
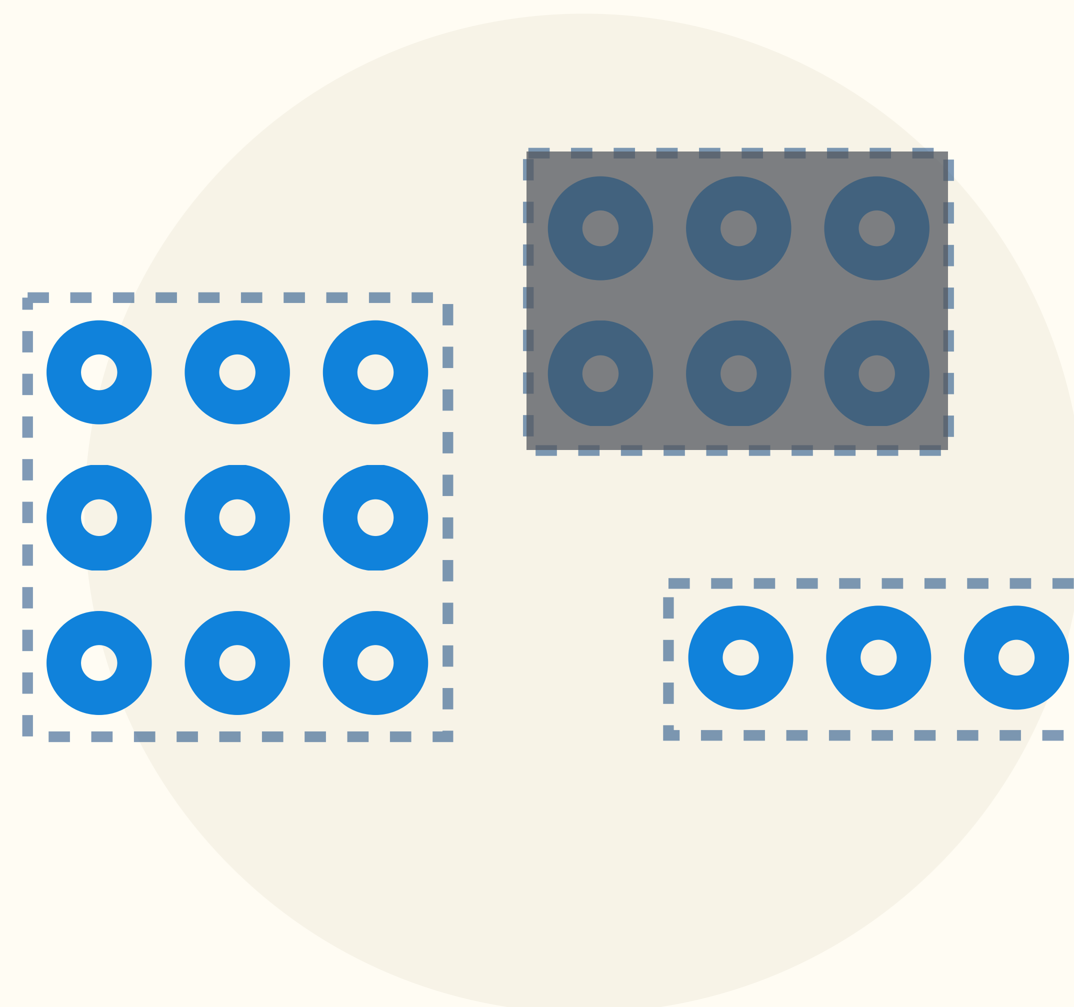
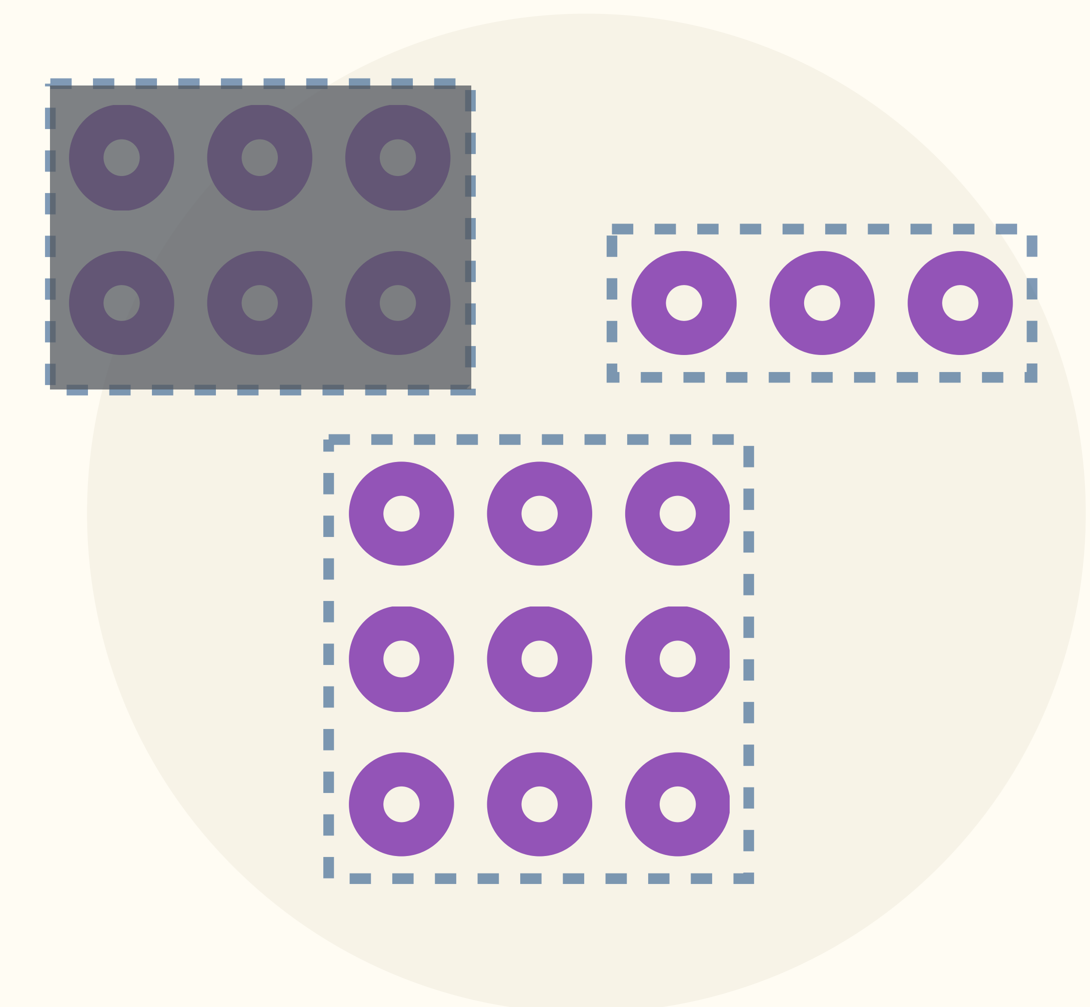




Avoid querying chunks via **constraint exclusion**

```
SELECT time, device_id, temp FROM data
   WHERE time > now() - interval '24 hours'
```

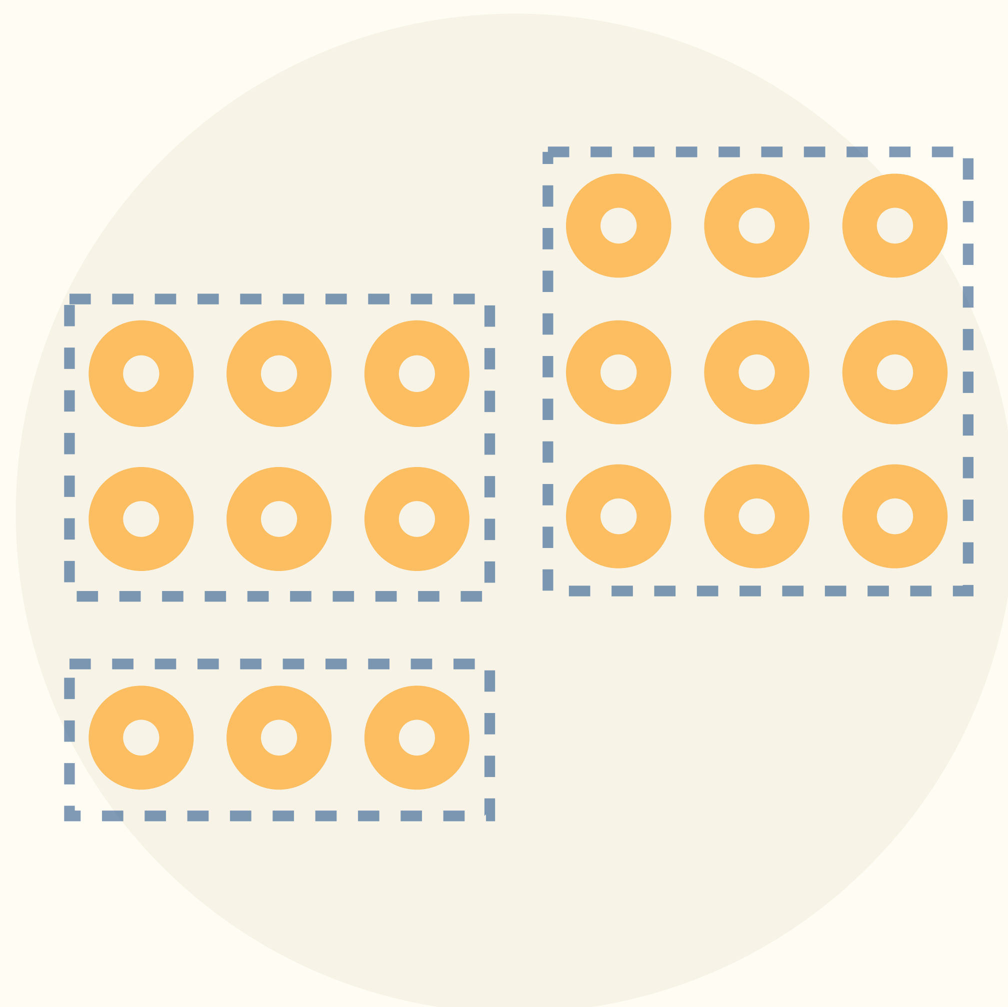
Plain Postgres won't exclude chunks

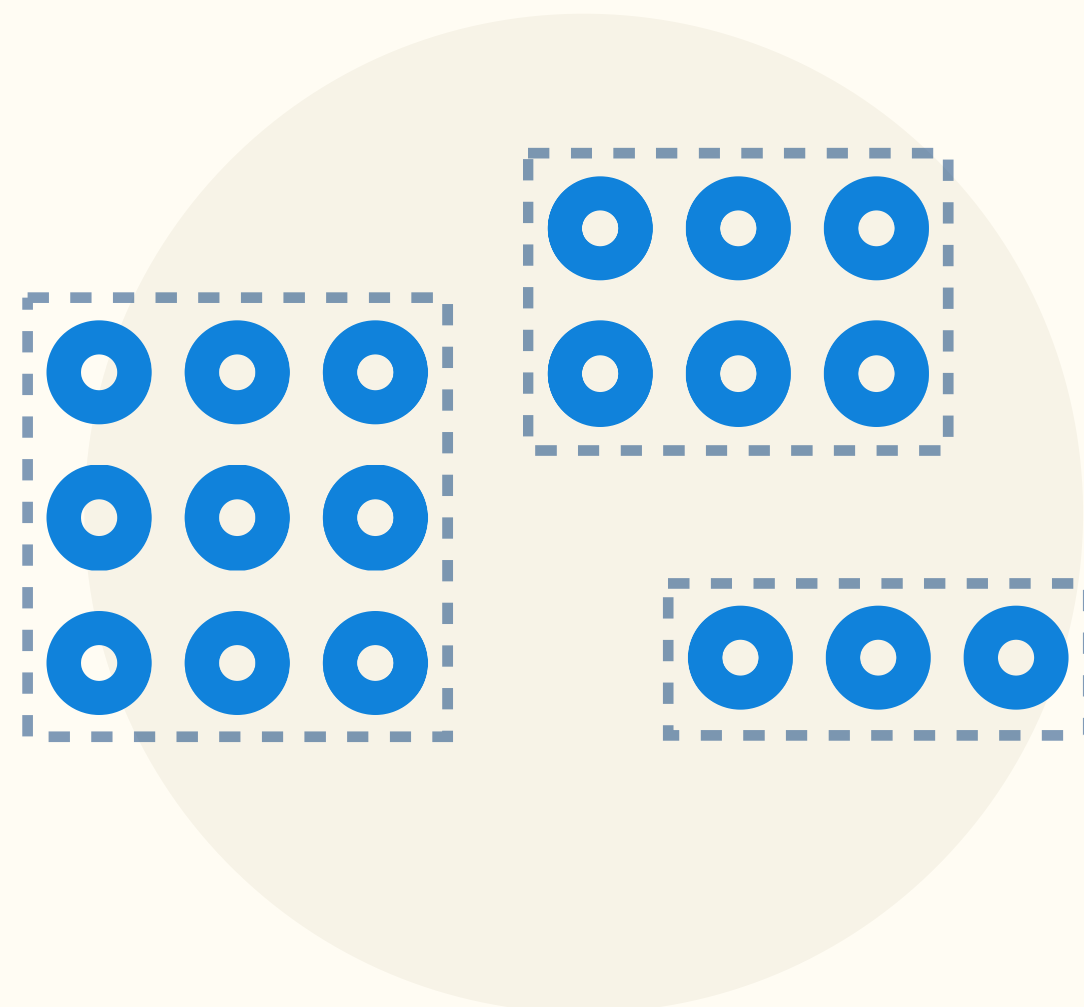




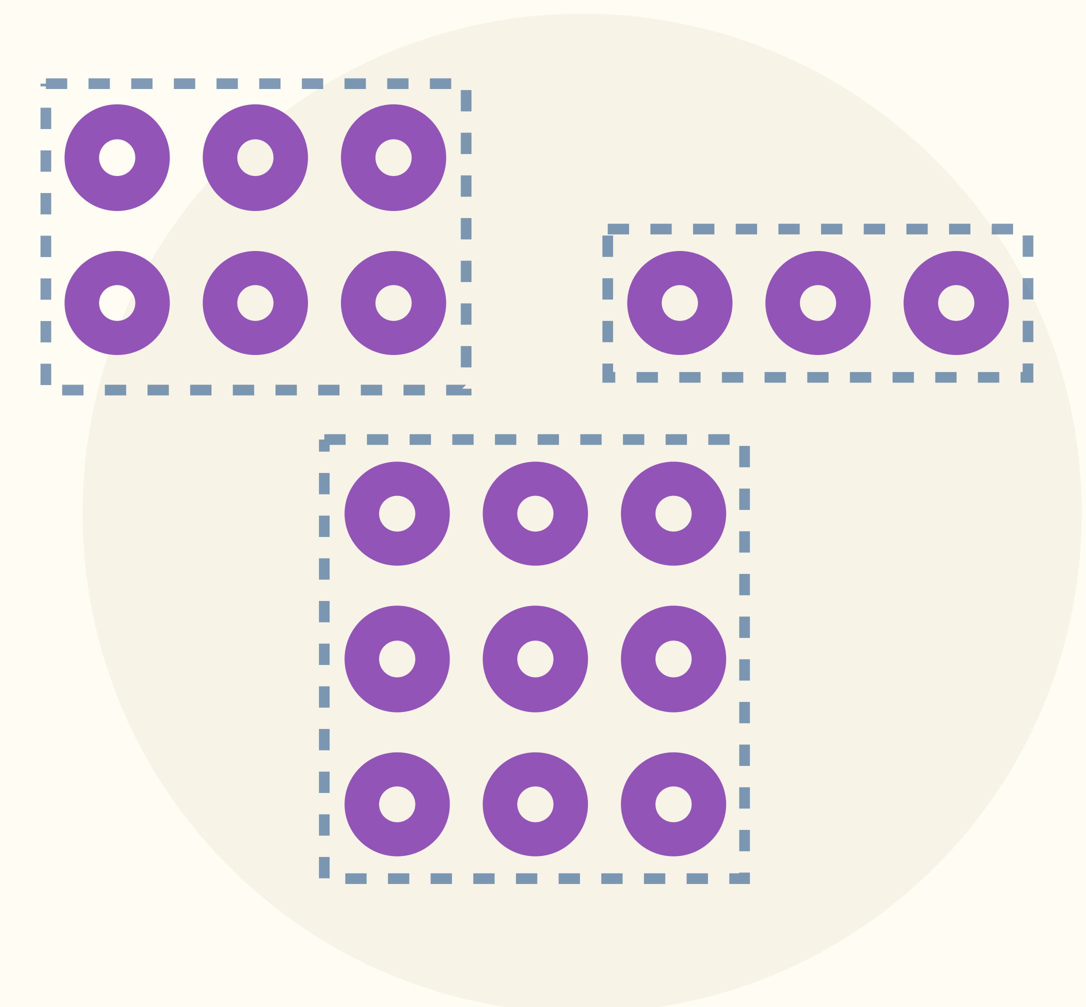
Efficient **retention policies**

Drop chunks, don't delete rows

⇒ avoids vacuuming







TimescaleDB vs. PostgreSQL

single-row inserts

144K
METRICS / S

PostgreSQL
TimescaleDB

Insert rate [rows / sec]

6,000
4,000
12,000
10,000
8,000
6,000
4,000
2,000
0

0 50 100 150 200 250

Dataset size [millions of rows]

14.4K
INSERTS / S

TimescaleDB 0.5, Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (LRS storage)

Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

TimescaleDB vs. PostgreSQL

batch inserts

1.11M METRICS / S

PostgreSQL
TimescaleDB

Insert rate [rows / s]

Dataset size [millions of rows]

>20x

TimescaleDB 0.5, Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (LRS storage)

Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

TimescaleDB vs. PostgreSQL

	SPEEDUP
Table scans, simple column rollups	**~0-20%**
GROUPBYs	**20-200%**
Time-ordered GROUPBYs	**400-10000x**
DELETEs	**2000x**

TimescaleDB 0.5, Postgres 9.6.2 on Azure standard DS4 v2 (8 cores), SSD (LRS storage)
Each row has 12 columns (1 timestamp, indexed 1 host ID, 10 metrics)

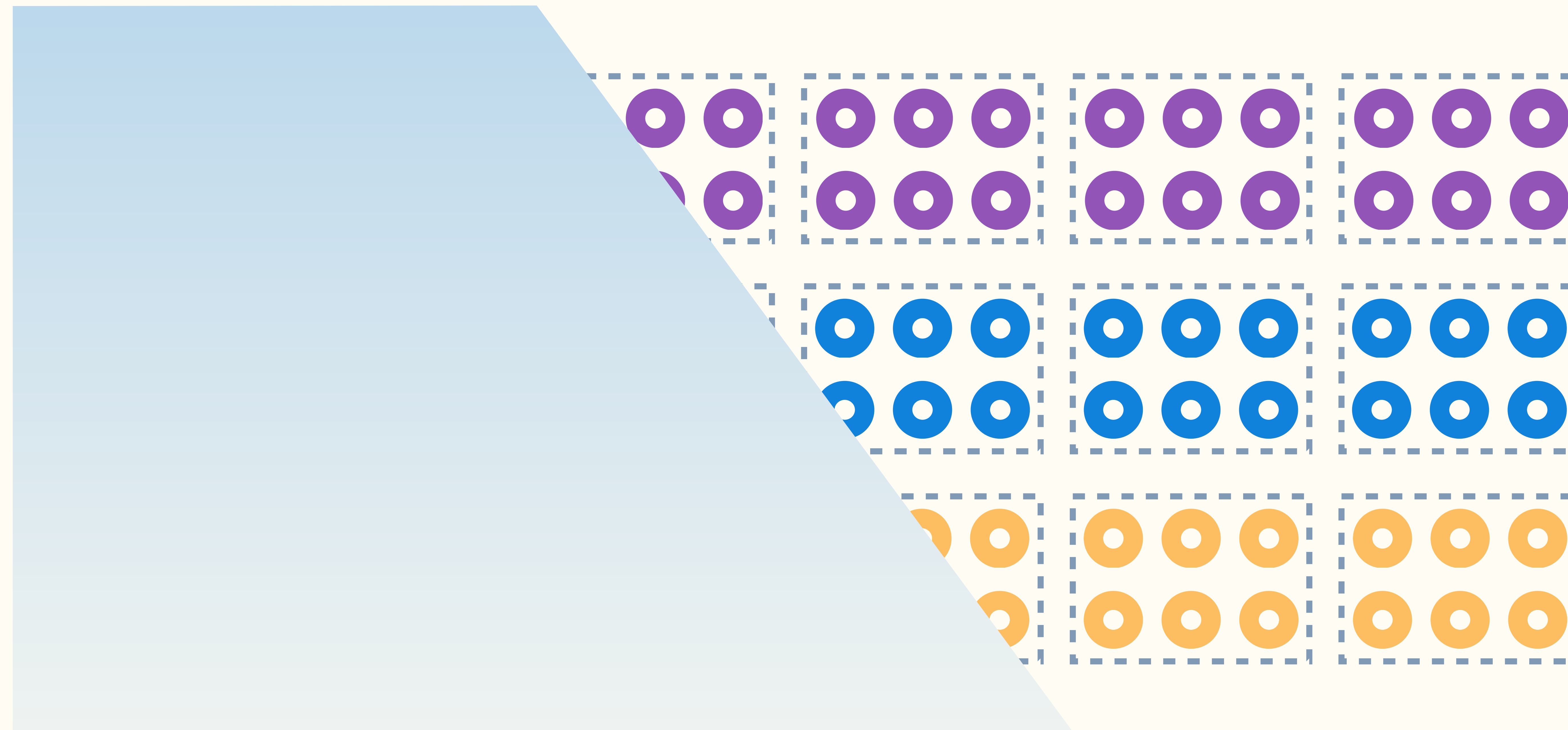
PostgreSQL 10 doesn’t solve the problem

(more later)

Not well supported in PG10

Hypertable

- Indexes
- Triggers
- Constraints
- UPSERTs
- Table mgmt



Chunks

TimescaleDB: Easy to Get Started

```
CREATE TABLE conditions (
      time timestamptz,
      temp float,
      humidity float,
      device text
);

SELECT create_hypertable('conditions', 'time', ‘device', 4,
      chunk_time_interval => interval '1 week’);

INSERT INTO conditions
      VALUES ('2017-10-03 10:23:54+01', 73.4, 40.7,
'sensor3');


SELECT * FROM conditions;

          time          | temp | humidity | device
------------------------+------+----------+---------
 2017-10-03 11:23:54+02 | 73.4 |     40.7 | sensor3
```

TimescaleDB: Repartitioning is Simple

```
— Set new chunk time interval
SELECT set_chunk_time_interval('conditions', interval '24 hours');

— Set new number of space partitions
SELECT set_number_partitions('conditions', 6);
```

```
CREATE TABLE conditions (
      time timestamptz,
      temp float,
      humidity float,
      device text
);

CREATE TABLE conditions_p1 PARTITION OF conditions
      FOR VALUES FROM (MINVALUE) TO ('g')
      PARTITION BY RANGE (time);
CREATE TABLE conditions_p2 PARTITION OF conditions
      FOR VALUES FROM ('g') TO ('n')
      PARTITION BY RANGE (time);
CREATE TABLE conditions_p3 PARTITION OF conditions
      FOR VALUES FROM ('n') TO ('t')
      PARTITION BY RANGE (time);
CREATE TABLE conditions_p4 PARTITION OF conditions
      FOR VALUES FROM ('t') TO (MAXVALUE)
      PARTITION BY RANGE (time);
-- Create time partitions for the first week in each device partition
CREATE TABLE conditions_p1_y2017m10w01 PARTITION OF conditions_p1
      FOR VALUES FROM ('2017-10-01') TO ('2017-10-07');
CREATE TABLE conditions_p2_y2017m10w01 PARTITION OF conditions_p2
      FOR VALUES FROM ('2017-10-01') TO ('2017-10-07');
CREATE TABLE conditions_p3_y2017m10w01 PARTITION OF conditions_p3
      FOR VALUES FROM ('2017-10-01') TO ('2017-10-07');
CREATE TABLE conditions_p4_y2017m10w01 PARTITION OF conditions_p4
      FOR VALUES FROM ('2017-10-01') TO (‘2017-10-07');

-- Create time-device index on each leaf partition
CREATE INDEX ON conditions_p1_y2017m10w01 (time);
CREATE INDEX ON conditions_p2_y2017m10w01 (time);
CREATE INDEX ON conditions_p3_y2017m10w01 (time);
CREATE INDEX ON conditions_p4_y2017m10w01 (time);

INSERT INTO conditions VALUES ('2017-10-03 10:23:54+01',
       73.4, 40.7, ‘sensor3');
```

PG10 requires a lot of **manual work**

500B
ROWS

400K
ROWS / SEC

50K
CHUNKS

5min
INTERVALS

*"As an update, we've scaled to beyond **500 billion rows now** and things are still working very smoothly.*

*The performance we're seeing is monstrous, almost **70% faster queries**!"*

- Software Engineer, Large Enterprise Computer Company

TimescaleDB vs. PostgreSQL 10

(single-row inserts)

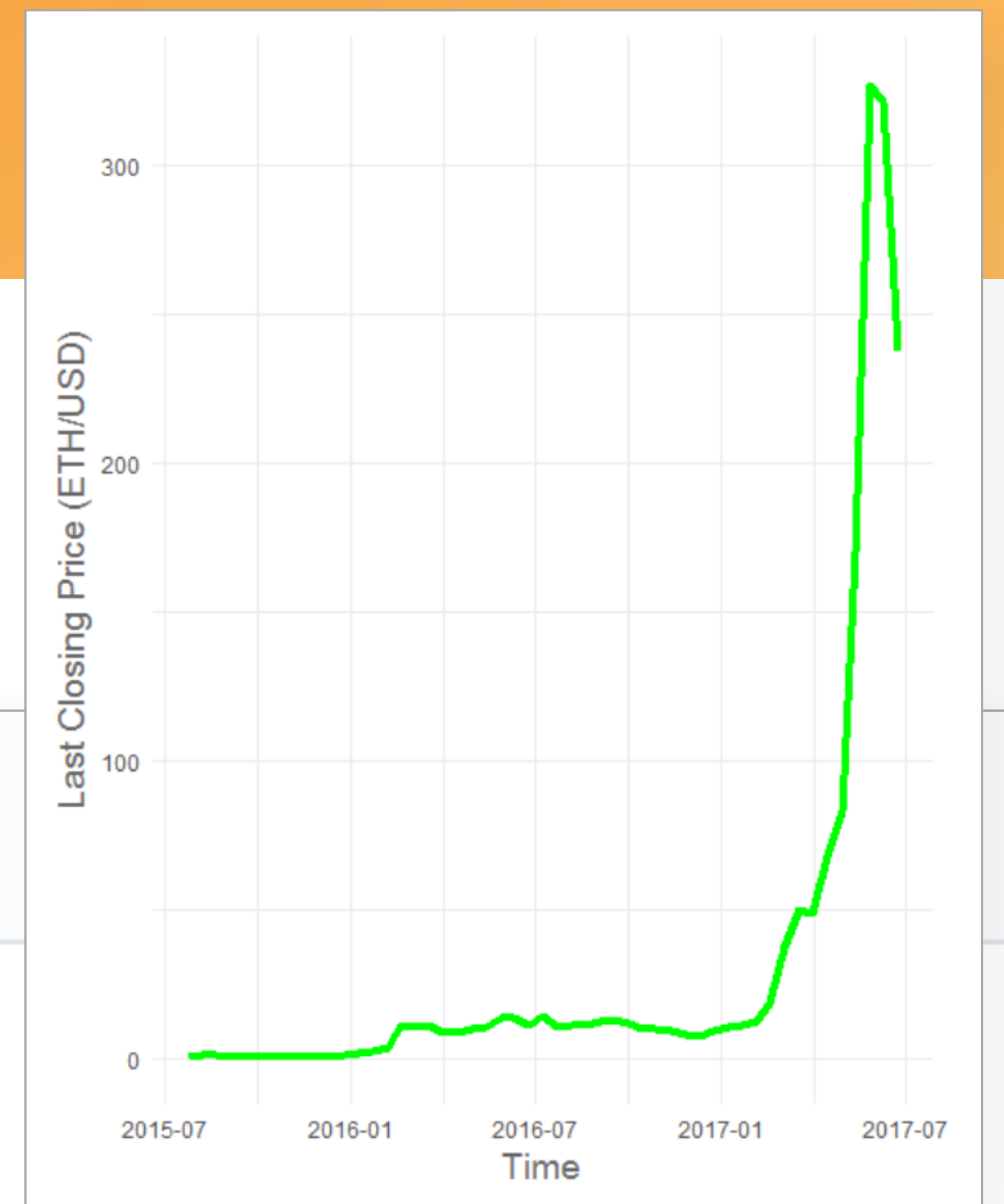
Example Query Optimization

```
CREATE INDEX ON readings(time);


SELECT    date_trunc('minute', time)          et,
          avg(cpu)
FROM      readings
GROUP BY  bucket
ORDER BY  bucket DESC
LIMIT 10;
```

Timescale understands time

Rich Time Analytics



eth_in_usd.sql

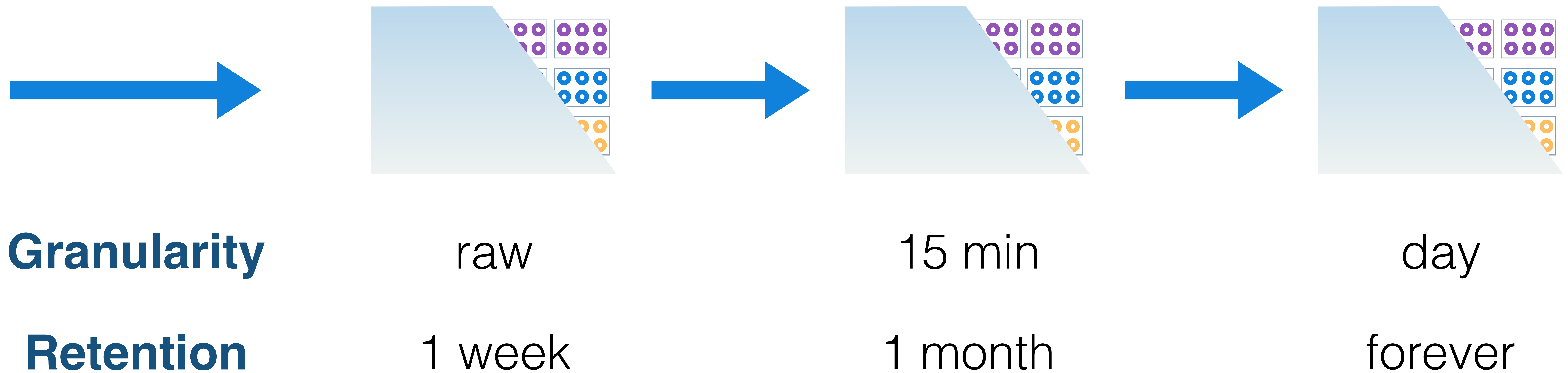
```
1   -- ETH prices in BTC and USD by two week intervals
2   SELECT time_bucket('14 days', c.time) as period,
3          last(c.closing_price, c.time) AS last_closing_price_in_btc,
4          last(c.closing_price, c.time) * last(b.closing_price, c.time) filter (WHERE b.currency
5   FROM crypto_prices c JOIN btc_prices b ON time_bucket('1 day', c.time) = time_bucket('1 day',
6   WHERE c.currency_code = 'ETH'
7   GROUP BY period
8   ORDER BY period;
```

Geospatial Temporal Analysis (with PostGIS)

trucks_tonnage_exiting.sql

```sql
SELECT time_bucket('1 day', time) as day,
       COUNT(*) as trucks_exiting, SUM(weight) as tonnage
  FROM vehicle_movements
  WHERE ST_Within(last_location,
      ST_Polygon((SELECT geom FROM cities WHERE name='los_angeles'),4326))
    AND NOT ST_Within(current_location,
      ST_Polygon((SELECT geom FROM cities WHERE name='los_angeles'),4326))
  GROUP BY day
  ORDER BY day DESC LIMIT 30;
```

Data Retention + Aggregations



Timescale is hiring!

- Core Database C Programmers
- R&D Engineers
- Solutions Engineers
- Evangelists
- Customer Success

timescale.com/careers

Other drivers for feasibility study

Comparison of data models

- Normalized (NM) → Invariant data extracted to other tables
- De-normalized (DNM) → Single table

Number of rows for scalability:

1M → 10M → 20M

Normalized model

HYPER TABLE

flux_data	
id	bigserial NOT NULL
time	bigint
ele_flux	double precision[]

Every ele_flux value is a matrix (NxM) where N and M may vary

	Energy1	Energy2	Energy3	…
Sector1	779.53	530.95	889.47	…
Sector2	422.20	267.98	871.70	…
Sector3	671.88	685.11	101.01	…
…	…	…	…	…

{{779.53,530.95,889.47,…},{422.20,267.98,871.70,…}, {671.88,685.11,101.01,…}…}

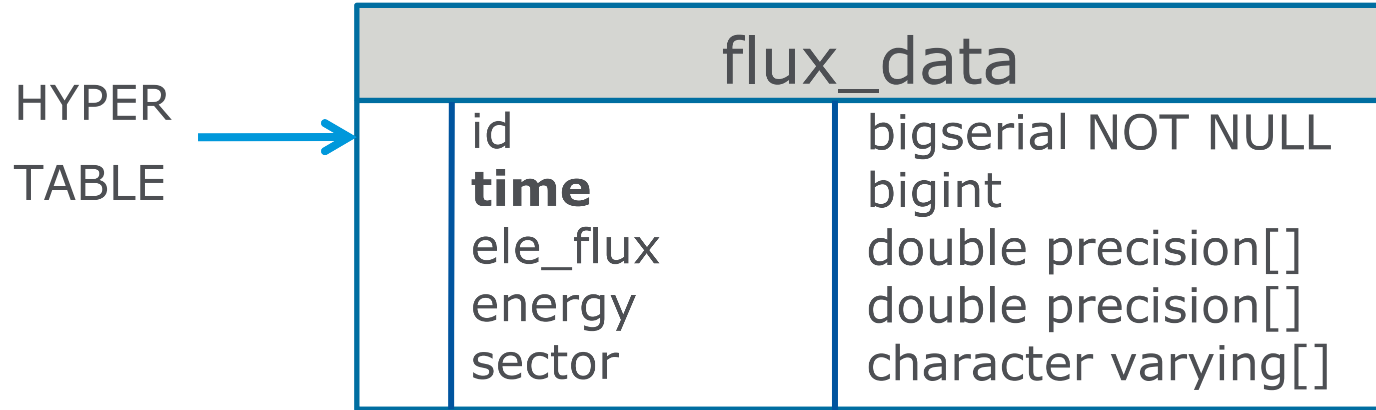
Normalized dependencies

flux_data	
id	bigserial NOT NULL
time	bigint
ele_flux	double precision[]

energy	
id	bigserial NOT NULL
value	double precision[]
start_time	bigint
end_time	bigint

sector	
id	bigserial NOT NULL
value	character varying[]

De-normalized Model



Environment

SERVER

- RHEL7
- Memory: 47 GB
- Local Disk: 218 GB
- Netapp: 497 GB

DATABASE

- PostgreSQL 10.1
- 2 Tablespaces
  - In Local → Run queries
  - In Netapp → Auxiliary for moving tables

Partitioning

- Size of partitions 1GB
- Time values increased consecutively one by one per row
- Hypertables → *chunk_time_interval = num_rows/table size in GB*
- Indexes on time column (default in Timescale)

Approach	Model	# Rows	Table Size	# Partitions
Pg10	Normalized	1/10/20 M	2/25/50 GB	2/25/50
Pg10	De-Normalized	1/10/20 M	10/41/99 GB	10/40/100
Timescale	Normalized	1/10/20 M	2/25/50 GB	3/25/50
Timescale	De-Normalized	1/10/20 M	10/41/99 GB	10/40/100

Queries – Final approach

TABLE TO QUERY | START/END TIME | START/END ENERGY

```
SELECT * FROM get_ele_flux (‘flux_data',4,6000000,3,37,
ARRAY['t1','m1','b1']);
```

SECTORS LIST

- Filters by range of time, energy and sectors dependencies
- Function which avoids calculation of array position per row

Function – First index of matrix

```
…
FOR sector_id IN (SELECT unnest(ARRAY['t1','m1','b1']) as value)
LOOP
        query_sectors:='select array_position(value, sector_id.value)
from sectors';
        EXECUTE query_sectors into sector_pos;
…
```

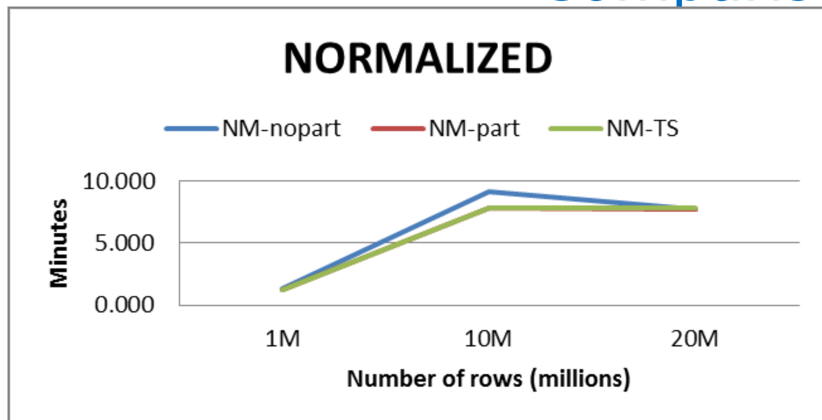
Function – Second index of matrix

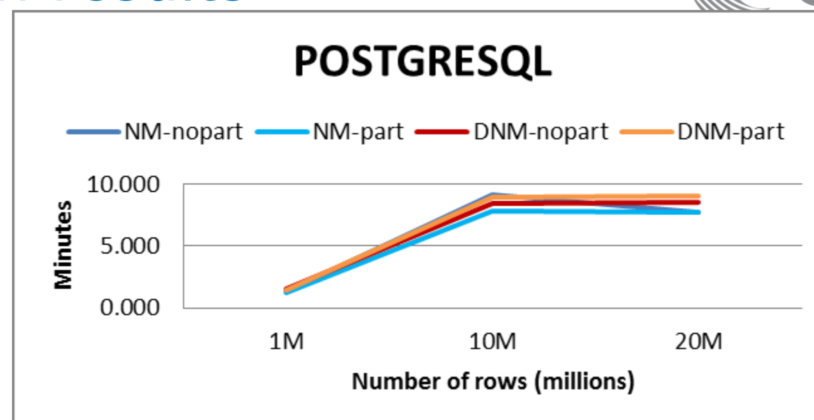
```
FOR record in (
select energy_value from (
   select unnest(value) as energy_value from energy where
   4<max_time and 6000000>min_time) as foo
   where energy_value >3 and energy_value <37
)
LOOP
     EXECUTE 'select array_position(value, record. energy_value) pos
from energy order by pos' into energy_pos;
```

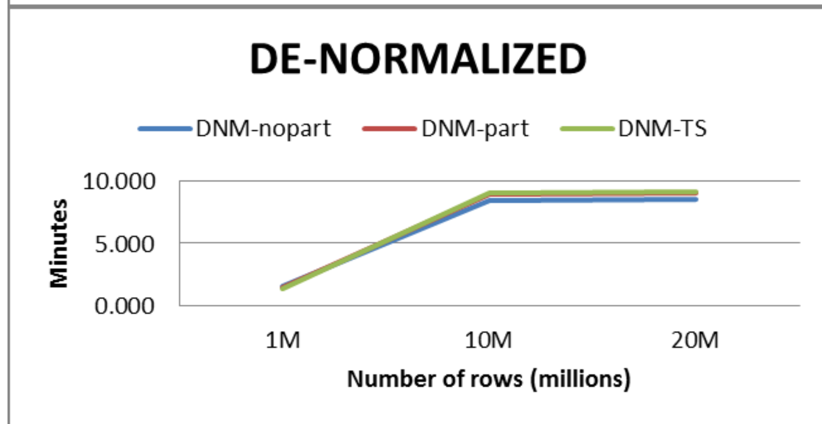
Function – Use indexes to get values

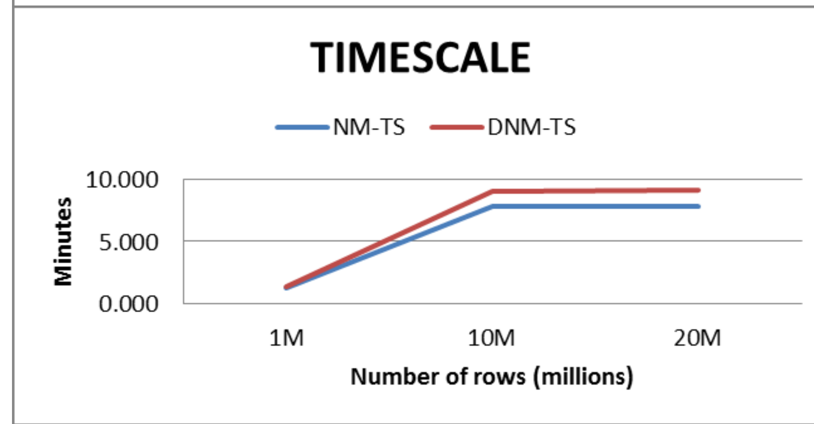
```
...
FOR record IN execute '(SELECT time,  ele_flux[sector_pos][energy_pos]
as ele_flux_value FROM flux_data where time>4 and time<6000000)'
LOOP
   time_value := record.time ;
   ele_flux_value := record.ele_flux_value;
   ...
   RETURN NEXT;
...
```

Comparison results

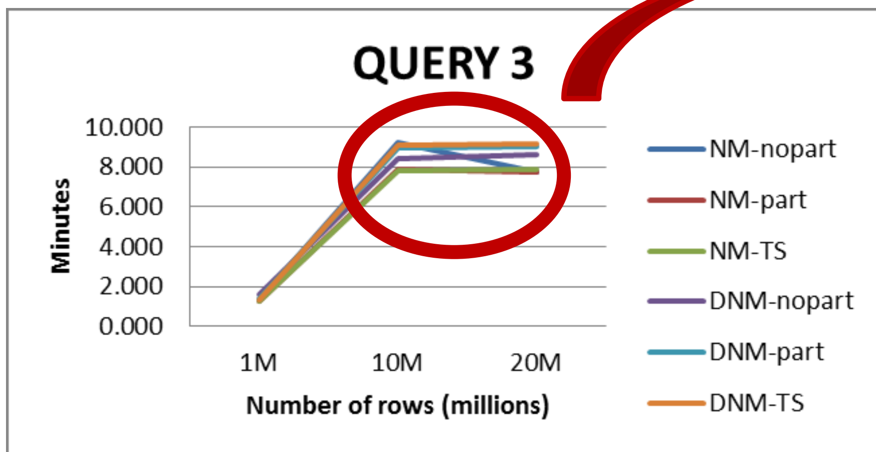


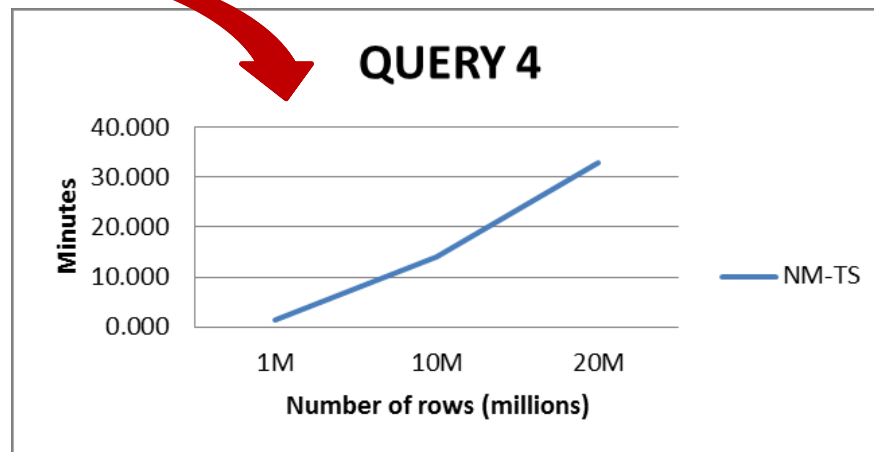






Scalability - Growth factor





Retrieved 6M 36M 36 M

Retrieved 6M 60M 120 M

- All rows scanned at all scenarios (unrealistic for time series)
- Growth factor → not good when raising to 100M rows retrieval

Conclusions for SOAR case

- Feasible to use RDBMS for SOAR time series data storage
  - Time series case → # of rows scanned/retrieved much lower
  - Coming performance improvements: Model, indexing, tiling, SSD disk for indexes, 1-2 TB RAM, parallel queries
- Using index on time, normalized model behaves better (15% gain)
- TimescaleDB eases the work
  - Similar performance for PG10 and TimescaleDB
  - Partition creation/maintenance effort saved with TimescaleDB
- Growth factor and UI constraints → Limit amount of points retrieved
  - Tiling, timeouts, user quotas, disk space quotas

What comes next?

- Working now on visualization of Scalar dataset and 1 dependency datasets
- Next steps
    1. Visualization of complex datasets in the User Interface
    2. Analyze/improve full model and indexing
    3. Data ingestion with on-the-fly hypertables creation
    4. Analysis and implementation of tiling algorithms

Thank you!

hector.perez@esa.int

david@timescale.com