Tracciare i cambiamenti in PostgreSQL con ClickHouse

Leonardo Cecchi

17 Maggio 2019

I database ed il fattore tempo

- Quante volte dobbiamo rispondere alla domanda: come mai quel record ha `is_active = False`? Chi lo può avere cambiato?

I database ed il fattore tempo

- Quante volte dobbiamo rispondere alla domanda: come mai quel record ha `is_active = False`? Chi lo può avere cambiato?
- created_at, created_by, updated_at, updated_by, ends_at, bastano davvero?

I database ed il fattore tempo

- Quante volte dobbiamo rispondere alla domanda: come mai quel record ha `is_active = False`? Chi lo può avere cambiato?
- created_at, created_by, updated_at, updated_by, ends_at, bastano davvero?
- E quante volte vorremmo avere la lista delle modifiche fatte nel nostro database per esaminarle? □

Dove mettiamo questi dati?

- Raccogliere i log in un folder basta davvero?

Dove mettiamo questi dati?

- Raccogliere i log in un folder basta davvero?
- Non sarebbe meglio avere uno strumento di analisi?

Dove mettiamo questi dati?

- Raccogliere i log in un folder basta davvero?
- Non sarebbe meglio avere uno strumento di analisi?
- Uno star-schema sarebbe appropriato per i log? □

PostgreSQL + WAL2JSON + FluentD + ClickHouse

Leonardo Cecchi


leonardo.cecchi@2ndquadrant.it


@leonardo_cecchi

DBMS: a quali domande risponde?

- Quanti clienti ho? OK!

DBMS: a quali domande risponde?

- Quanti clienti ho? OK!
- Mario Rossi è un mio cliente? OK!

DBMS: a quali domande risponde?

- Quanti clienti ho? OK!
- Mario Rossi è un mio cliente? OK!
- Quanti clienti ho in ciascuna provincia? OK!

DBMS: a quali domande risponde?

- Quanti clienti ho? OK!
- Mario Rossi è un mio cliente? OK!
- Quanti clienti ho in ciascuna provincia? OK!
- `Leonardo Cecchi` era un mio cliente nel 03-01-1999?
  dipende

Closed World Assumption

CWA (Closed World Assumption) la relazione contiene **tutti** i predicati veri. Ad es. `mario rossi` è un mio cliente solo se è compreso nella tabella del **clienti**.

Non ci sono dubbi, ma solo **NULL**i!

Il fattore tempo

`Leonardo Cecchi` era un mio cliente nell'11-01-2019?

Aggiungiamo i campi:

- `starts_at: date`
- `ends_at: date`

Basteranno?

Esempio

- Leonardo Cecchi viene inserito come cliente nel 09-01-2019;
- chiede la rimozione dall'elenco dei clienti nel 18-01-2019;
- nel 23-01-2019 viene iscritto nuovamente nell'elenco

Nome	Cognome	Starts	Ends
Leonardo	Cecchi	09-01	18-01
Leonardo	Cecchi	23-01	+infty

Normalizzazione

Nome	Cognome	Starts	Ends
Leonardo	Cecchi	09-01	18-01
Leonardo	Cecchi	23-01	+infty

- Creare una tabella per i nomi, ed una per gli intervalli di validità
- Rinunciare a normalizzare i campi

Dal punto di vista operazionale

Nome	Cognome	Starts	Ends
Leonardo	Cecchi	09-01	18-01
Leonardo	Cecchi	23-01	+infty

- ogni modifica è un inserimento (MVCC)
- qualche volta la lista dei cambiamenti è tanto importante quanto il risultato finale (ad esempio è importante sapere quando e perché il gruppo sanguigno di un paziente è stato cambiato da 0+ a 0−)?

MVCC

- nessun dato viene mai cambiato, i dati vecchi vengono marcati come “non visibili”;
- progressivamente il loro spazio verrà riusato da dati freschi;
- PostgreSQL era pensato per gestire una gerarchia di storage:
  - shared buffers: molto efficiente, ma volatile;
  - storage persistente (dischi): mediamente efficiente e persistente, da usare per conservare gli shared buffers;
  - memorie meno efficienti (tape): da usare per memorizzare i dati che sono stati marcati per il riuso, ma utili per rispondere a query temporali. *Pianificato per Ingres ma mai realizzato*

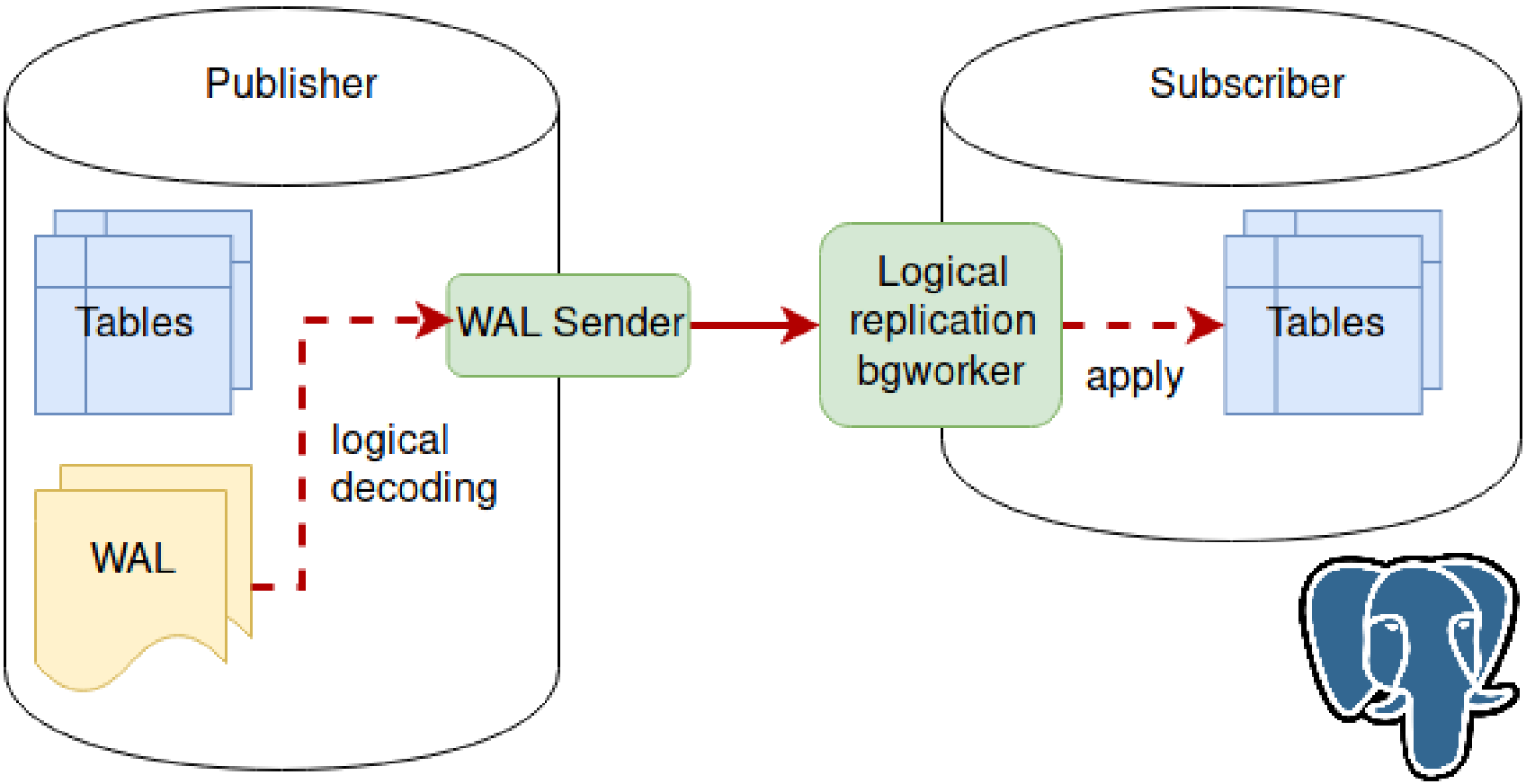
(Readings in Database Systems, 5th Edition, http://www.redbook.io/)

Architettura di PostgreSQL

Due database in uno:

- situazione corrente dell'area dati (shared buffers & storage);
- flusso dei cambiamenti all'area dati (WAL)
  - pensato per crash recovery;
  - utilizzato per backup fisici;
  - utilizzato per replica fisica;
  - utilizzato per replica logica.

Replica logica in PostgreSQL



Come decodificare il flusso delle modifiche? WAL2JSON

- wal2json è un **logical decoding plugin**;
- ogni transazione accettata viene serializzata in JSON:

```
{
 "xid":583,
 "timestamp": "2018-03-27 11:58:28.98",
 "change": [
   {
     "kind": "insert",
     "schema": "public",
     "table": "table_with_pk",
     "columnnames": ["a", "b", "c"],
     "columnvalues": [1, "...", "2018-03-27 11:58:28.98"]
   }
 ]
}
```

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica
- Open Source (Apache v2)

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica
- Open Source (Apache v2)
- SQL

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica
- Open Source (Apache v2)
- SQL
- Column Store

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica
- Open Source (Apache v2)
- SQL
- Column Store
- Massively Parallel Processing

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica
- Open Source (Apache v2)
- SQL
- Column Store
- Massively Parallel Processing
- Linux!

Dove mettiamo i dati? ClickHouse

Cosa è ClickHouse?

- Analytical database per large dataset creato da Yandex per il servizio Metrica
- Open Source (Apache v2)
- SQL
- Column Store
- Massively Parallel Processing
- Linux!
- Mostly Append-Only

ClickHouse: Users

CLOUDFLARE®

ClickHouse: Architettura

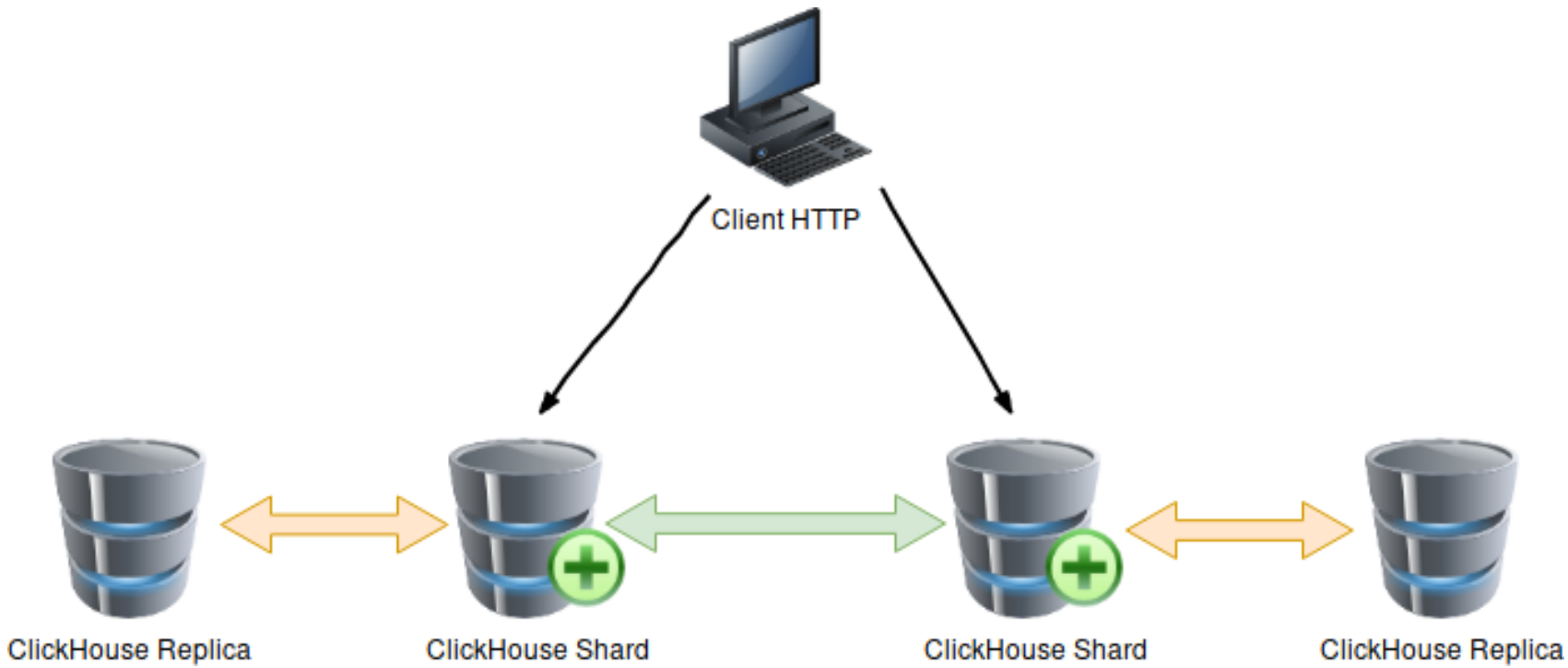


Figure 1: Architettura tipica

Clickhouse: Quando usarlo?

- La maggior parte delle query leggono dati

Clickhouse: Quando usarlo?

- La maggior parte delle query leggono dati
- I dati vengono aggiornati in batch grandi (1000 righe)

Clickhouse: Quando usarlo?

- La maggior parte delle query leggono dati
- I dati vengono aggiornati in batch grandi (1000 righe)
- Le letture estraggono molte righe dal database, ma con solo un sottoinsieme delle colonne

Clickhouse: Quando usarlo?

- La maggior parte delle query leggono dati
- I dati vengono aggiornati in batch grandi (1000 righe)
- Le letture estraggono molte righe dal database, ma con solo un sottoinsieme delle colonne
- Le transazioni non sono necessarie

Clickhouse: Quando usarlo?

- La maggior parte delle query leggono dati
- I dati vengono aggiornati in batch grandi (1000 righe)
- Le letture estraggono molte righe dal database, ma con solo un sottoinsieme delle colonne
- Le transazioni non sono necessarie
- Il risultato delle query può essere allocato agilmente in RAM

ClickHouse: Performance

```
:) select count(*) from logs_postgresql format Vertical;

SELECT count(*)
FROM logs_postgresql
FORMAT Vertical

Row 1:
------
count(): 72768780

1 rows in set. Elapsed: 0.057 sec.
Processed 72.77 million rows, 291.08 MB
(1.29 billion rows/s., 5.14 GB/s.)
```

ClickHouse: Performance (2)

```
:) SELECT
    count(*),
    min(length(column_values)),
    max(length(column_values))
FROM logs_postgresql
WHERE table_name = 'pgbench_accounts'
FORMAT Vertical

count():                        18192195
min(length(column_values)): 99
max(length(column_values)): 110

1 rows in set. Elapsed: 1.233 sec.
Processed 18.19 million rows, 2.58 GB
(14.75 million rows/s., 2.09 GB/s.)
```

ClickHouse: Performance (1)

```
:) select count(*) from dw.ad8_fact_event;

SELECT count(*)
FROM dw.ad8_fact_event

┌─────count()─┐
│ 1261705085657 │
└───────────────┘

1 rows in set. Elapsed: 3.552 sec. Processed 1.26 trillion rows, 1.26 TB (355.22 billion
rows/s., 355.22 GB/s.)
```

Figure 2: Sequential query example

(Benchmark eseguito da Alexander Zaitsev di Altinity e usato con il suo permesso)

ClickHouse: Performance (2)

```
:) select sum(price_cpm) from dw.ad8_fact_event where access_day=today()-1 and event_key=-2;

SELECT sum(price_cpm)
FROM dw.ad8_fact_event
WHERE (access_day = (today() - 1)) AND (event_key = -2)

┌─sum(price_cpm)─┐
│ 87579.09035192338 │
└──────────────────┘

1 rows in set. Elapsed: 0.168 sec. Processed 161.89 million rows, 2.91 GB (961.83 million
rows/s., 17.31 GB/s.)
```

Figure 3: Fragmented query example

(Benchmark eseguito da Alexander Zaitsev di Altinity e usato con il suo permesso)

ClickHouse: MergeTree

È il metodo preferito di memorizzare moli di dati in ClickHouse:

- i dati vengono velocemente scritti in una parte
- parti piccole vengono fuse e ordinate secondo una chiave scelta
- due parti sono unibili solo se appartengono alla stessa partizione

ClickHouse: Partizionamento orizzontale

```
CREATE TABLE logs_postgresql (
        event_ts DateTime,
        receive_ts DateTime,
        xid Int64,
        kind String,
        schema_name String,
        table_name String,
        column_names Nullable(String),
        column_values Nullable(String),
        [...]
)
ENGINE MergeTree
PARTITION BY toYYYYMM(event_ts)
ORDER BY event_ts;
```

ClickHouse: MergeTree family

In realtà gli indici MergeTree sono una famiglia:

- ReplacingMergeTree per deduplicare i dati
- SummingMergeTree per sommare i dati che hanno la stessa chiave
- AggregatingMergeTree per l'aggregazione generica (min, max, count)
- CollapsingMergeTree per la gestione del "segno"

ClickHouse: Mutations

- Mostly Append-Only

ClickHouse: Mutations

- Mostly Append-Only
- ALTER TABLE `name` DELETE WHERE `cond`
- ALTER TABLE `name` UPDATE `col` = `expr` WHERE `cond`

ClickHouse: Mutations

- Mostly Append-Only
- ALTER TABLE `name` DELETE WHERE `cond`
- ALTER TABLE `name` UPDATE `col` = `expr` WHERE `cond`
- ALTER TABLE `name` [DETACH|ATTACH] PARTITION `expr`
- ALTER TABLE `name` DROP PARTITION `expr`
- ALTER TABLE `name` FREEZE PARTITION `expr` □

ClickHouse: Indici sparsi

MergeTree ammette un solo indice (lo chiama *chiave primaria*).

Ogni 8192 record viene enunciato il valore e puntato il relativo record.

ClickHouse: Distributed Engine

- Ogni server nel cluster ha una partizione dei dati
- L'albero di esecuzione viene diviso fra i server
- Il server che ha proposto la query raccoglie i risultati intermedi e li combina. Attenzione alla dimensione dello stato intermedio!

ClickHouse: Stato intermedio e funzioni approssimanti

- `uniqExact`: le occorrenze univoche devono essere trasmesse fra i server
- `uniqCombined`: solo un filtro di Bloom passa

ClickHouse: ReplicatedMergeTree engine

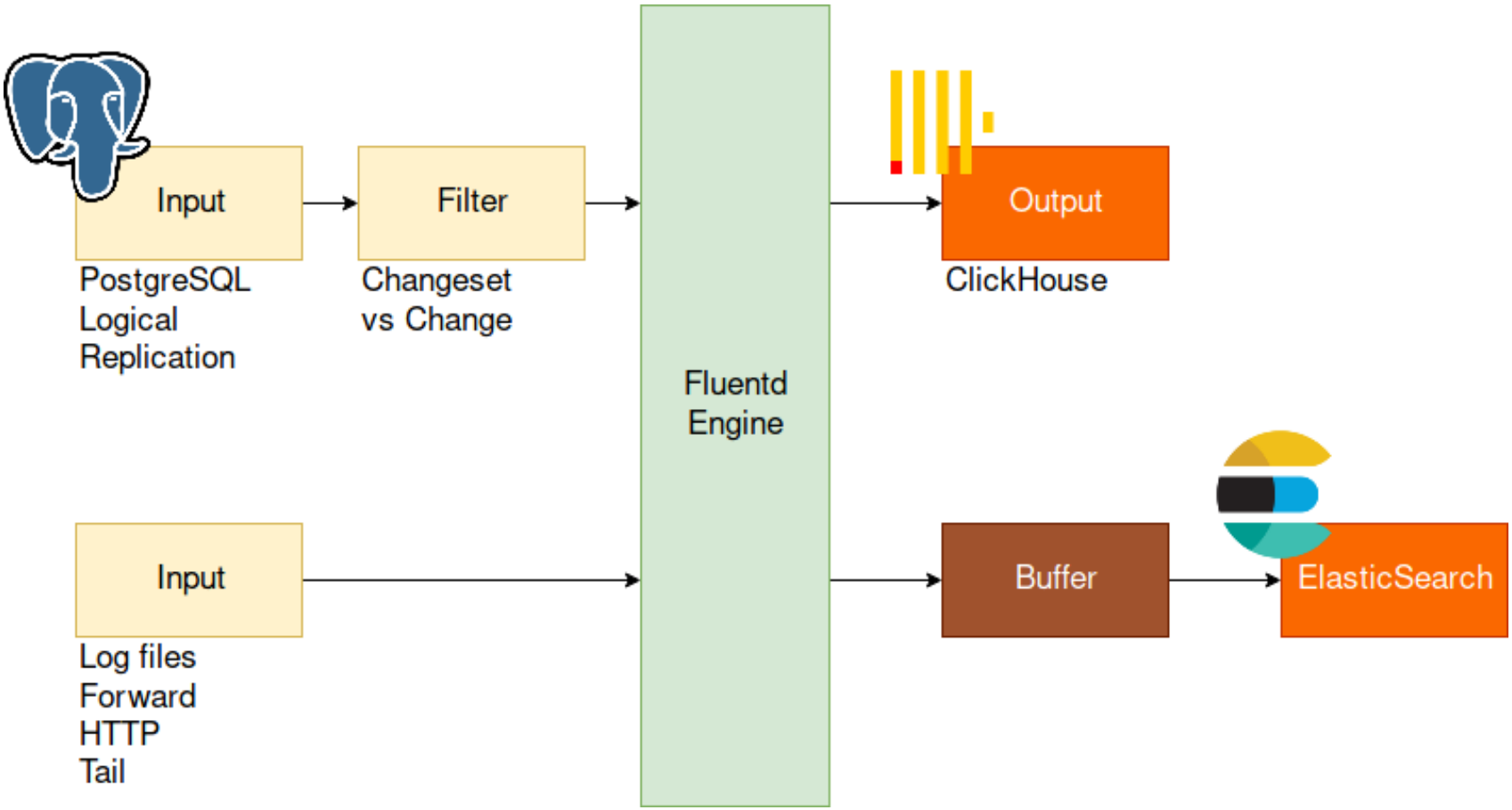
- Tutti i server del cluster contengono gli stessi dati
- Sincronizzazione attraverso ZooKeeper

Il piano:

- PostgreSQL espone il flusso delle modifiche in JSON, usando WAL2JSON
- FluentD (log collector)
  - legge le modifiche in formato JSON usando `pg_recvlogical`
  - trasforma da un JSON per ChangeSet ad un JSON per transazione
  - inserisce le modifiche in ClickHouse

Si può fare!

Piano dei lavori



FluentD - Input

```
<source>
  @type exec
  command pg_recvlogical [...]
  format json
  tag postgres.changes
</source>
```

pg_recvlogical

Si aggancia ad un server PostgreSQL utilizzando il protocollo della replica logica, attivando il plugin di decodifica e scrivendo i dati su stdout:

```
pg_recvlogical
   --start
   --slot changes_slot
   -P wal2json
   -f -
   -o [...]
   -d postgres
```

Opzioni WAL2JSON

- `include-xids=true`
- `include-timestamp=true`

  Inserimento dello XID e del timestamp nel JSON

- `write-in-chunks=true`

  Molto importante: permette di fare **flush ad ogni cambiamento** e non ad ogni changeset. Permette a FluentD di non andare in OOM su transazioni molto grandi.

Da changeset a change

```
<filter postgres.*>
 @type pg_logical_rows
</filter>
```

Filtro custom

```
module Fluent::Plugin
  class PostgresLogicalRowsFilter < Filter
    Fluent::Plugin.register_filter(
      'pg_logical_rows', self)

    def filter_stream(tag, es)
      [...]
      new_es
    end
  end
end
```

Filtro custom

```
new_es = Fluent::MultiEventStream.new
es.each {|time, record|
  record["change"].each {|change|
    target = {}
    target["receive_ts"] = time
    target["xid"] = record["xid"]

    if not change["columnnames"].nil?
      [...]
      target["column_values"] = [...]
    end
    [...]
    new_es.add(time, target)
  }
}
```

Output

```
<match postgres.*>
  @type exec
  command /usr/local/bin/insert_ch.sh
  format json
  buffer_type memory
  buffer_chunk_limit 64m
  buffer_queue_limit 32
  flush_at_shutdown true
  flush_interval 5s
  num_threads 4
</match>
```

Memorizzare il change in ClickHouse

ClickHouse permette di inserire dati JSON in modo diretto.

```
cat $1 | clickhouse-client
  --host=[...]
  --compression true
  --query="INSERT INTO logs_postgresql
           FORMAT JSONEachRow"
```

Risultati:

```
:) select event_ts, xid, kind,
table_name, column_names, column_values
from logs_postgresql
order by event_ts desc
limit 5 format Vertical;

Row 1:
------
event_ts:      2019-05-05 11:23:34
xid:           14721333
kind:          update
table_name:    pgbench_branches
column_names:  "bid" "bbalance" "filler"
column_values: "8" "-7947682" ""
```

Risultati e performance

```
:) select count(*) from logs_postgresql format Vertical;

Row 1:
──────
count(): 38160112

1 rows in set. Elapsed: 0.050 sec.
Processed 38.16 million rows, 152.64 MB
(766.29 million rows/s., 3.07 GB/s.)
```

Risultati e performance

```
:) select schema_name, table_name, count(*)
from logs_postgresql
group by schema_name, table_name
format PrettySpace;

schema_name   table_name          count()

public        pgbench_accounts    9896010
public        pgbench_history     9890350
public        pgbench_tellers     9896010
public        pgbench_branches    9896010

4 rows in set. Elapsed: 3.334 sec.
Processed 39.58 million rows, 1.56 GB
(11.87 million rows/s., 468.94 MB/s.)
```

Controlli da fare

- Ma PostgreSQL avrà subito qualche impatto?
- Chi è il collo di bottiglia?

Impatto WAL2JSON

Macchina virtuale, 2 CPU non riservate, 4GB RAM.

- Senza WAL2JSON e collector: **1051** TPS
- Con WAL2JSON: **964** TPS

Impatto di circa 8% del numero massimo di TPS erogabili

Miglioramenti successivi

La CPU del server FluentD è utilizzata per intero:

- Usare una batteria di FluentD anziché un solo processo
- Evitare il filtro (nuova versione di WAL2JSON)
- Implementare il filtro in modo più efficiente

Grazie!

Leonardo Cecchi


leonardo.cecchi@2ndquadrant.it


@leonardo_cecchi

We are hiring!