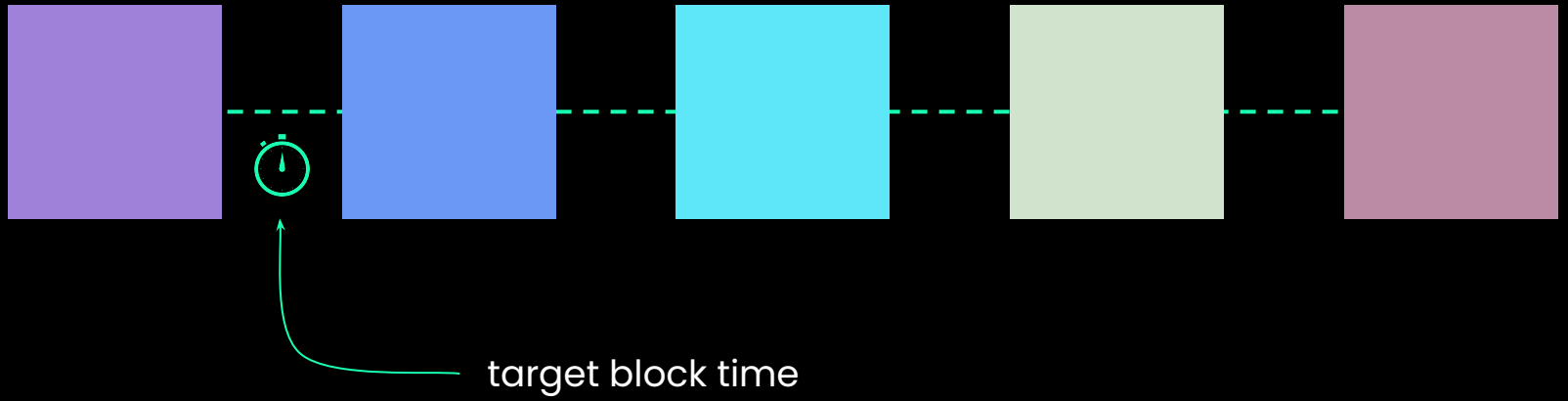
Benchmarking Deep Dive

Shawn Tabrizi
Software Engineer
Parity Technologies

Overview

- Quick Recap of Weights
- Deep Dive Into Benchmarking
- Our Learnings Throughout Development
- Best Practices and Common Patterns

What is Weight?

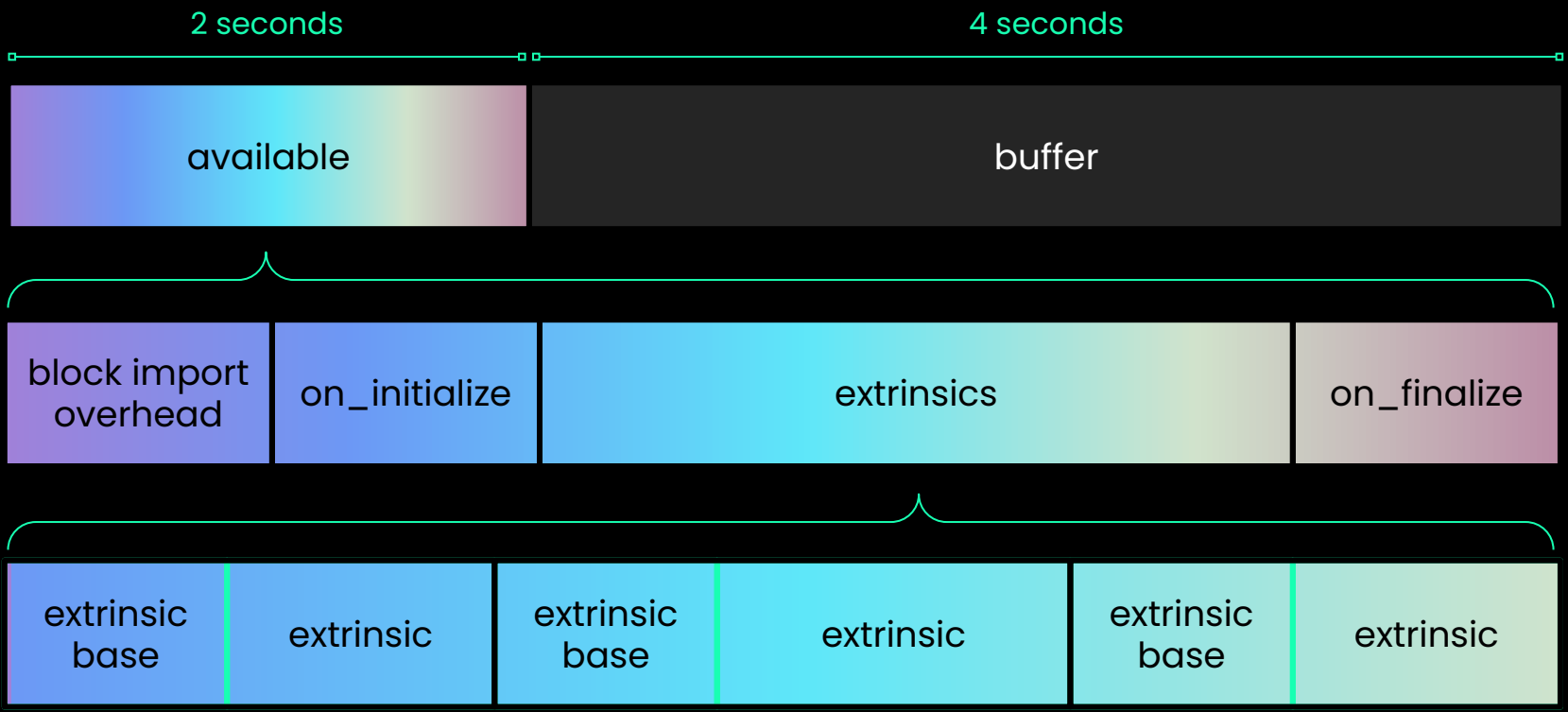


Weight is a representation of block execution time.

10^{12} Weight = 1 Second

i.e. 1,000 weight = 1 nanosecond

Block Import Weight Breakdown



Weight is specific to each blockchain.

- 1 second of compute on different computers allows for different amounts of computation.
- Weights of your blockchain **will** evolve over time.
- Higher hardware requirements will result in a more performant blockchain (i.e. TXs per second), but will limit the kinds of validators that can safely participate in your network.

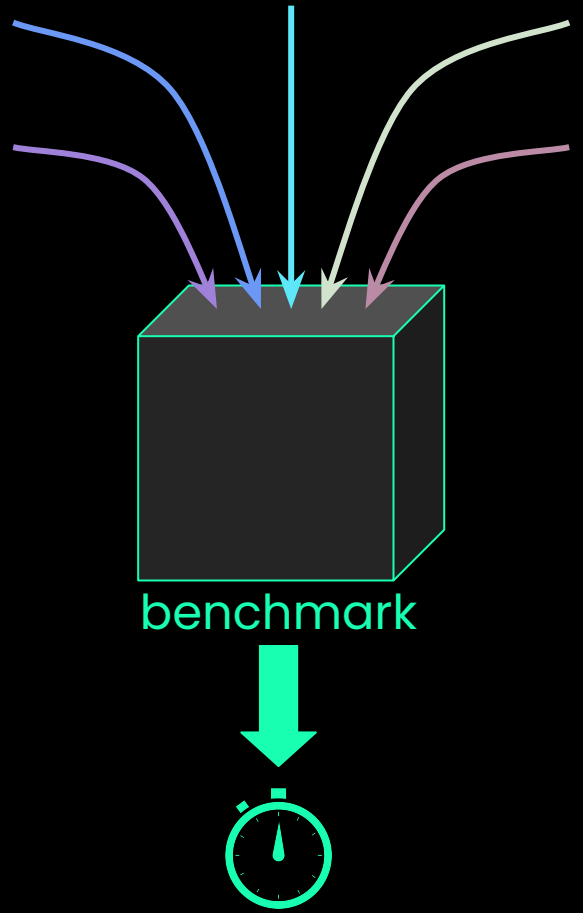
ASSUMPTIONS

- Processor
- Memory
- Hard Drive
  - HDD vs. SSD vs. NVME
- Operating System
- Drivers
- Rust Compiler
- Runtime Execution Engine
  - compiled vs. interpreted
- Database
  - RocksDB vs. ParityDB vs. ?

The Benchmarking Framework

The Benchmarking Plan

- Use **empirical** measurements of the runtime to determine the time it takes to execute extrinsics and other runtime logic.
- Run benchmarks using worst case scenario conditions.
    - Primary goal is to keep the runtime safe.
    - Secondary goal is to be as accurate as possible to maximize throughput.

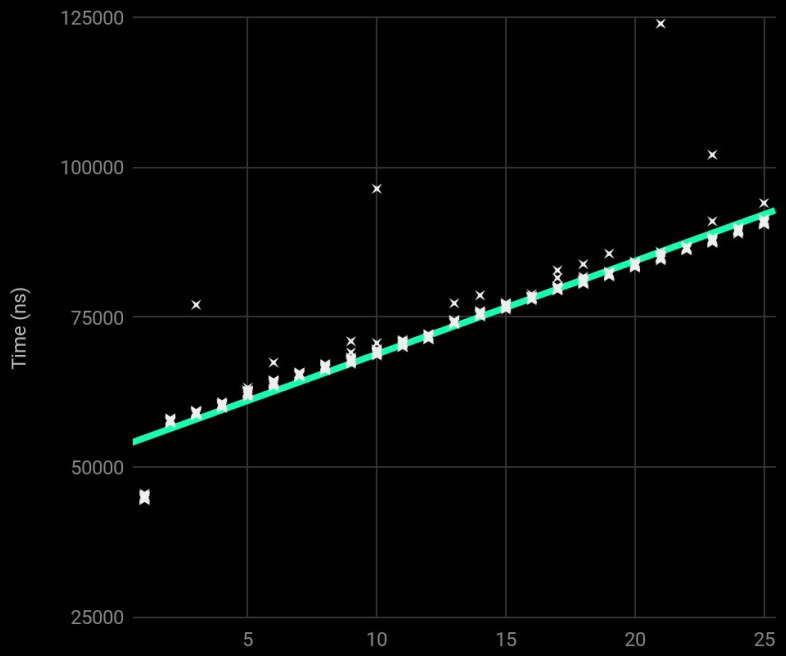


Reference: https://www.shawntabrizi.com/substrate/substrate-weight-and-fees/

The benchmarks! Macro

```
benchmarks! {
    extrinsic_name {
        /* setup initial state */
    }: _{ /* execute extrinsic or function */ }
    verify {
        /* verify final state */
    }
}
```

Multiple Linear Regression Analysis



- We require that no functions in Substrate have superlinear complexity.
- Ordinary least squared linear regression.
  - `linregress` crate
- Supports multiple linear coefficients.
  - $Y = Ax + By + Cz + k$
- For constant time functions, we simply use the median value.

The benchmark CLI

```
➜  substrate benchmark --help

substrate-benchmark 2.0.0

Benchmark runtime pallets.



USAGE:

    substrate benchmark [FLAGS]
[OPTIONS] --extrinsic <extrinsic>
--pallet <pallet>
```

- Compile your node with:

  `--features runtime-benchmarks`
- Use subcommand “benchmark”
- Feedback wanted!
  - Expect a lot of continuous iteration here…

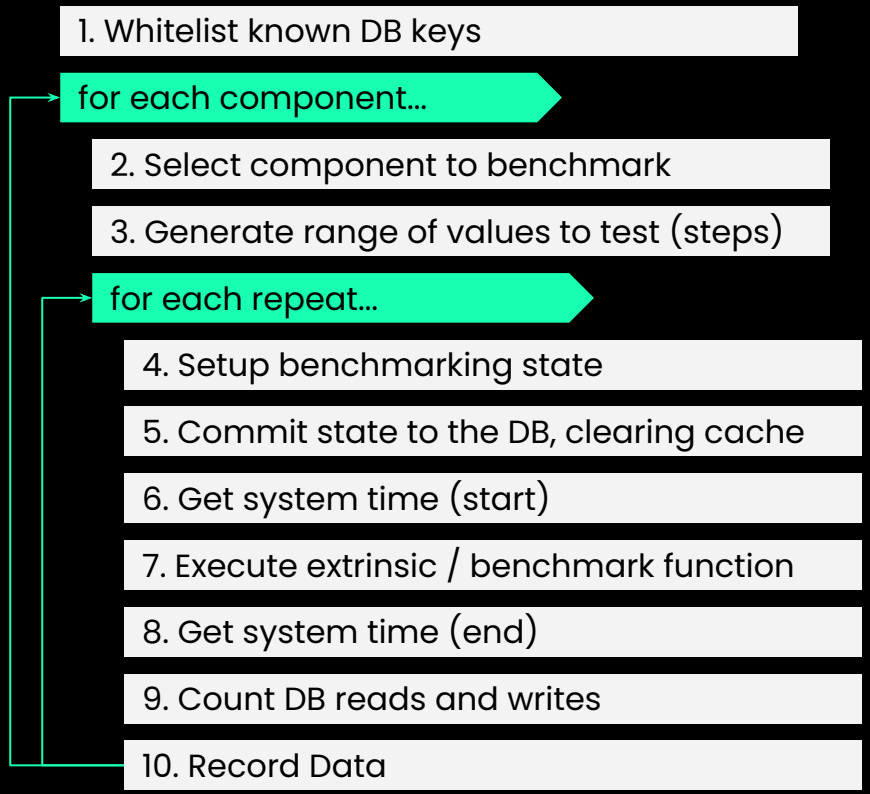
Autogenerated WeightInfo

```
pub trait WeightInfo {
    fn transfer() -> Weight;
    fn transfer_keep_alive() -> Weight;
    fn set_balance_creating() -> Weight;
    fn set_balance_killing() -> Weight;
    fn force_transfer() -> Weight;
}
```

- Final output is a Rust file.
- Implements `WeightInfo` trait, unique to each pallet.
- These functions are used to define the weight of their respective extrinsic.
- Completely configurable if desired.
- Designed to make end to end benchmark automation easy.

Deep Dive

The Benchmarking Process



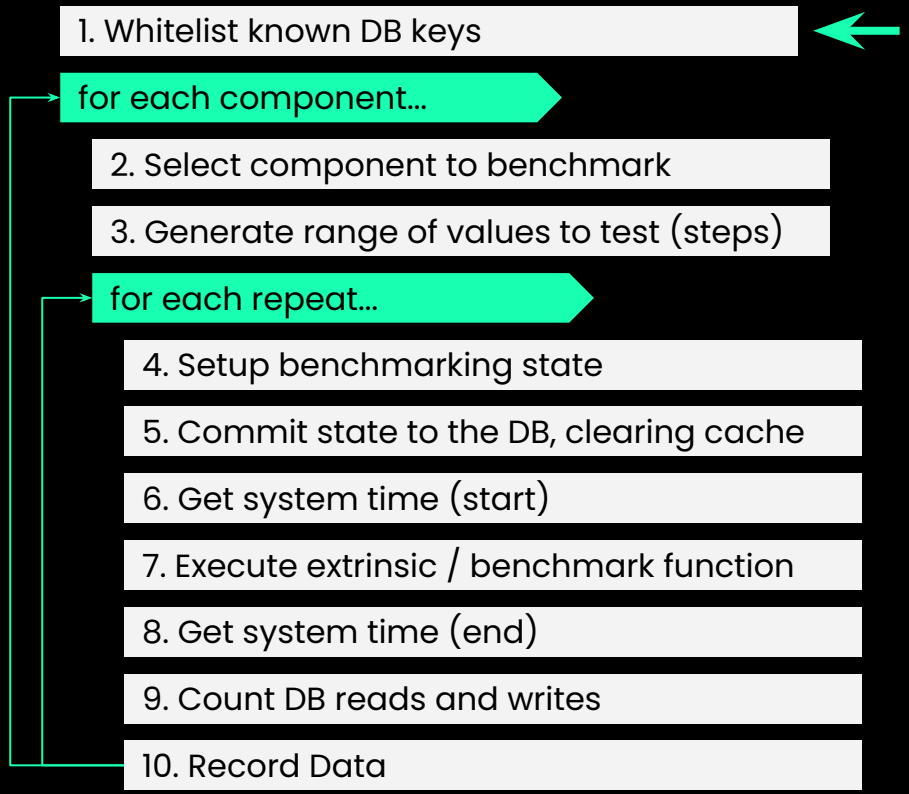
So let's walk through the steps of a benchmark!

Reference:

`frame/benchmarking/src/lib.rs`

`-> fn run_benchmark(...)`

The Benchmarking Process



- Some keys are accessed every block:
  - Block Number
  - Events
  - Total Issuance
  - etc...
- We don't want to count these reads and writes in our benchmarking results.
- Defined in `runtime/src/lib.rs`
- Applied to all benchmarks being run.
- This includes a "whitelisted account".

The Benchmarking Process

1. Whitelist known DB keys
- for each component...
  - 2. Select component to benchmark
  - 3. Generate range of values to test (steps)
  - for each repeat...
    - 4. Setup benchmarking state
    - 5. Commit state to the DB, clearing cache
    - 6. Get system time (start)
    - 7. Execute extrinsic / benchmark function
    - 8. Get system time (end)
    - 9. Count DB reads and writes
    - 10. Record Data

- Imagine a function with 3 components
  - let x in 0..5;
  - let y in 0..10;
  - let z in 0..1;
- We set number of steps to 3.
- Vary one component at a time, select high value for the others.
- Result:

x	0	1	2	3	4	5	5	5	5	5	5	5
y	10	10	10	10	10	10	0	3	6	9	10	10
z	1	1	1	1	1	1	1	1	1	1	0	1

The Benchmarking Process

1. Whitelist known DB keys
- for each component...
  - 2. Select component to benchmark
  - 3. Generate range of values to test (steps)
  - for each repeat...
    - 4. Setup benchmarking state
    - 5. Commit state to the DB, clearing cache
    - 6. Get system time (start)
    - 7. Execute extrinsic / benchmark function
    - 8. Get system time (end)
    - 9. Count DB reads and writes
    - 10. Record Data

5 / 3 -> Step Size 1

10 / 3 -> Step Size 3

1 / 3 -> Step Size 1

x	0	1	2	3	4	5	5	5	5	5	5	5
y	10	10	10	10	10	10	0	3	6	9	10	10
z	1	1	1	1	1	1	1	1	1	1	0	1

The Benchmarking Process

1. Whitelist known DB keys

for each component...

2. Select component to benchmark

3. Generate range of values to test (steps)

for each repeat...

4. Setup benchmarking state

5. Commit state to the DB, clearing cache

6. Get system time (start)

7. Execute extrinsic / benchmark function

8. Get system time (end)

9. Count DB reads and writes

10. Record Data

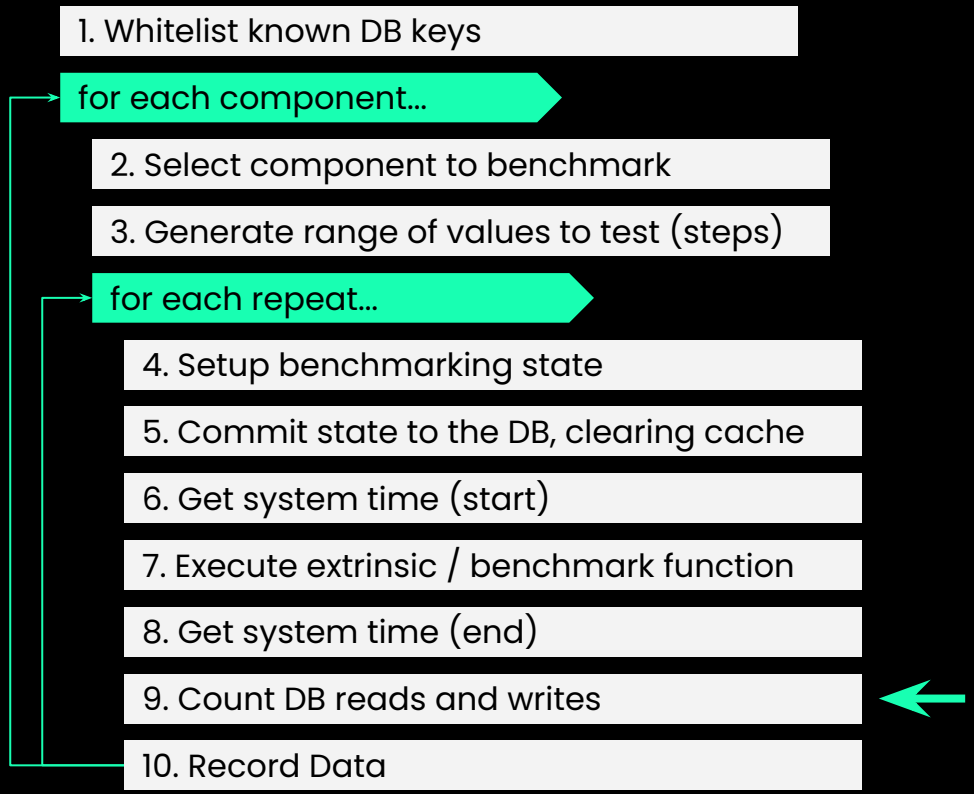
```
benchmarks! {
    extrinsic_name {
        /* setup initial state */
    }: _{ /* execute extrinsic */ }
    verify {
        /* verify final state */
    }
}
```

The Benchmarking Process

1. Whitelist known DB keys
- for each component...
  2. Select component to benchmark
  3. Generate range of values to test (steps)
  - for each repeat...
    4. Setup benchmarking state
    5. Commit state to the DB, clearing cache
    6. Get system time (start)
    7. Execute extrinsic / benchmark function
    8. Get system time (end)
    9. Count DB reads and writes
    10. Record Data

```
benchmarks! {
    extrinsic_name {
        /* setup initial state */
    }: _{ /* execute extrinsic */ }
    verify {
        /* verify final state */
    }
}
```

The Benchmarking Process



- We only care about the items going between the overlay storage and the merkle trie.
    - i.e. first read and first write to a unique storage key.
- We assume overlay storage is equivalent overhead as in-memory DB, which is already used for benchmarking.
- First write to a storage key counts as first read too.

The Benchmarking Process

1. Whitelist known DB keys

for each component...

2. Select component to benchmark

3. Generate range of values to test (steps)

for each repeat...

4. Setup benchmarking state

5. Commit state to the DB, clearing cache

6. Get system time (start)

7. Execute extrinsic / benchmark function

8. Get system time (end)

9. Count DB reads and writes

10. Record Data

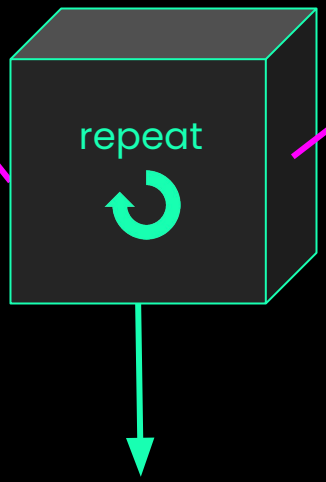
```rust
pub struct BenchmarkBatch {
    pub pallet: Vec<u8>,
    pub benchmark: Vec<u8>,
    pub results: Vec<BenchmarkResults>,
}

pub struct BenchmarkResults {
    pub components: Vec<(BenchmarkParameter, u32)>,
    pub extrinsic_time: u128,
    pub storage_root_time: u128,
    pub reads: u32,
    pub repeat_reads: u32,
    pub writes: u32,
    pub repeat_writes: u32,
}
```

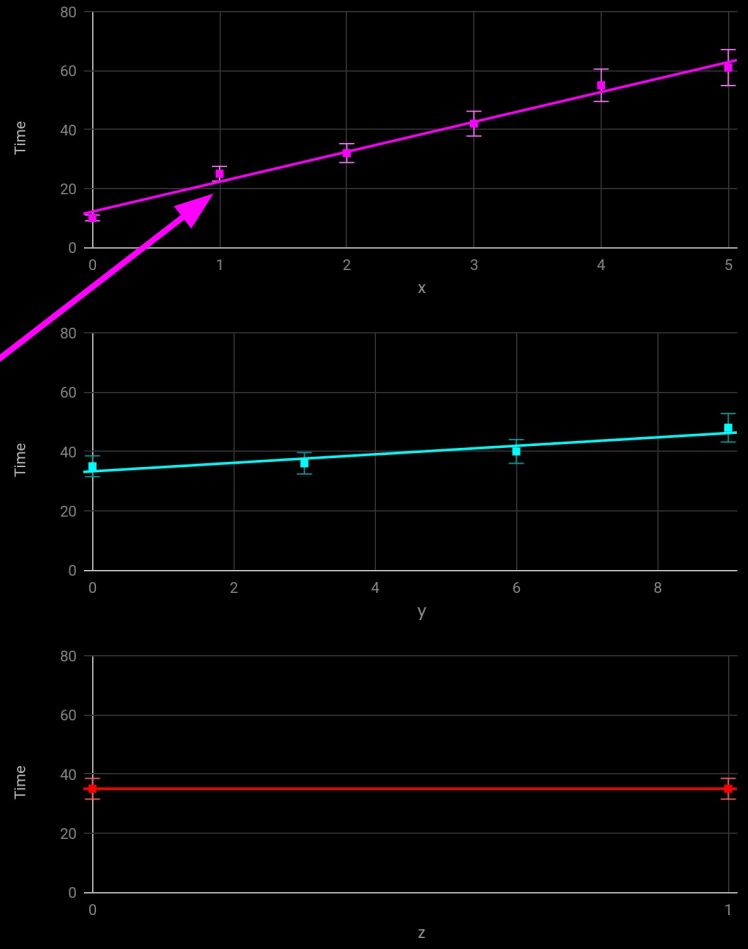
Components		
x	y	z
0	10	1
1	10	1
2	10	1
3	10	1
4	10	1
5	10	1
5	0	1
5	3	1
5	6	1
5	9	1
5	10	0
5	10	1

benchmark

repeat



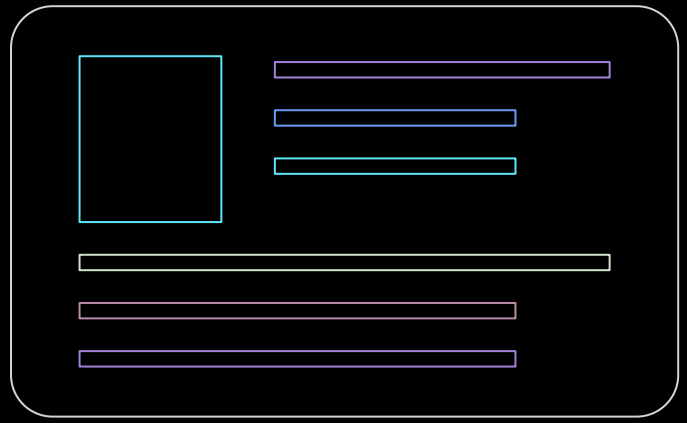
time = Ax + By + Cz + K



Example Benchmark

The Identity Pallet

Quick Review: Identity Pallet



- Identity can have variable amount of information
  - Name
  - Email
  - Twitter
  - etc...
- Identity can be judged by a variable amount of registrars.
- Identity can have a two-way link to “sub-identities”
  - Other accounts that inherit the identity status of the “super-identity”

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
    let target = T::Lookup::lookup(target)?;

    let (subs_deposit, sub_ids) = <SubsOf<T>>::take(&target);
    let id = <IdentityOf<T>>::take(&target).ok_or(Error::<T>::NotNamed)?;
    let deposit = id.total_deposit() + subs_deposit;
    for sub in sub_ids.iter() {
        <SuperOf<T>>::remove(sub);
    }

    T::Slashed::on_unbalanced(T::Currency::slash_reserved(&target,
deposit).0);

    Self::deposit_event(RawEvent::IdentityKilled(target, deposit));
}
```

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
```

- Our extrinsic uses a configurable origin.
- Our benchmark needs to always work independent of the configuration.
- We added a special function behind a feature flag:

```
    /// Returns an outer origin capable of passing `try_origin` check.
    ///
    /// ** Should be used for benchmarking only!!! **
    #[cfg(feature = "runtime-benchmarks")]
    fn successful_origin() -> OuterOrigin;
```

```
}
```

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
    let target = T::Lookup::lookup(target)?;
```

- In general, hooks like these are configurable in the runtime.
- Each blockchain will have their own logic, and thus their own weight.
- We run benchmarks against the real runtime, so we get the real results.
- **IMPORTANT!** You need to be careful that the limitations of these hooks are well understood by the pallet developer and users of your pallet, otherwise, your benchmark will not be accurate.

```
}
```

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
    let target = T::Lookup::lookup(target)?;

    let (subs_deposit, sub_ids) = <SubsOf<T>>::take(&target);
    let id = <IdentityOf<T>>::take(&target).ok_or(Error::<T>::NotNamed)?;
```

- 2 storage reads and writes.
- The size of these storage items will depends on:
  - Number of Registrars
  - Number of Additional Fields

```
eposit).0);

}
```

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
    let target = T::Lookup::lookup(target)?;

    let (subs_deposit, sub_ids) = <SubsOf<T>>::take(&target);
    let id = <IdentityOf<T>>::take(&target).ok_or(Error::<T>::NotNamed)?;
    let deposit = id.total_deposit() + subs_deposit;
    for sub in sub_ids.iter() {
        <SuperOf<T>>::remove(sub);
    }
                                                                    eposit).0);
}
```

- S storage writes, where S is the number of sub-accounts.
- S is unknown for the initial weight, so we must assume an upper bound.

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
    let target = T::Lookup::lookup(target)?;

    T::Slashed::on_unbalanced(T::Currency::slash_reserved(&target, deposit).0);

    Self::deposit_event(RawEvent::IdentityKilled(target, deposit));
}
```

- A balance operation, generally takes 1 read and 1 write
  - But is also configurable depending on where you store balances!
- What happens with slashed funds is configurable too!

Extrinsic: Kill Identity

```
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) {
    T::ForceOrigin::ensure_origin(origin)?;
    let target = T::Lookup::lookup(target)?;

    let (subs_deposit, sub_ids) = <SubsOf<T>>::take(&target);
    let id = <IdentityOf<T>>::take(&target).ok_or(Error::<T>::NotNamed)?;
    let deposit = id.total_deposit() + subs_deposit;
    for sub in sub_ids.iter() {
        <SuperOf<T>>::remove(sub);
    }
    T::Slashed::on_unbalanced(T::Currency::slash_reserved(&target, deposit).0);
    Self::deposit_event(RawEvent::IdentityKilled(target, deposit));
}
```

- We whitelist changes to the Events storage item, so generally this is “free” beyond computation and in-memory DB weight.

Benchmark: Kill Identity

```
kill_identity {
    let r in 1 .. T::MaxRegistrars::get();
    let s in 1 .. T::MaxSubAccounts::get();
    let x in 1 .. T::MaxAdditionalFields::get();

    let target: T::AccountId = account("target", SEED, SEED);
    let target_origin: <T as frame_system::Trait>::Origin = RawOrigin::Signed(target.clone()).into();
    let target_lookup: <T::Lookup as StaticLookup>::Source = T::Lookup::unlookup(target.clone());
    let _ = T::Currency::make_free_balance_be(&target, BalanceOf::<T>::max_value());

    let info = create_identity_info::<T>(x);
    Identity::<T>::set_identity(target_origin.clone(), info)?;
    let _ = add_sub_accounts::<T>(&target, s)?;

    // User requests judgement from all the registrars, and they approve
    add_registrars::<T>(r)?
    for i in 0..r {
        Identity::<T>::request_judgement(target_origin.clone(), i, 10.into())?;
        Identity::<T>::provide_judgement( /* snip */ )?;
    }

    let call = Call::<T>::kill_identity(target_lookup);
    let force_origin = T::ForceOrigin::successful_origin();
    ensure!(IdentityOf::<T>::contains_key(&target), "Identity not set");
}: { call.dispatch_bypass_filter(force_origin)? }
verify {
    ensure!(!IdentityOf::<T>::contains_key(&target), "Identity not removed");
}
```

Benchmark: Kill Identity

```
kill_identity {
    let r in 1 .. T::MaxRegistrars::get();
    let s in 1 .. T::MaxSubAccounts::get();
    let x in 1 .. T::MaxAdditionalFields::get();
```

- Our components.
  - R = Number of Registrars
  - S = Number of Sub-Accounts
  - X = Number of Additional Fields on the Identity.
- Note all of these have configurable, known at compile time maxima.
  - Part of the pallet configuration trait.
  - Runtime logic should enforce these limits.

```
    ensure!(IdentityOf::<T>::contains_key(&target), "Identity not set");
}: { call.dispatch_bypass_filter(force_origin)? }
verify {
    ensure!(!IdentityOf::<T>::contains_key(&target), "Identity not removed");
}
```

Benchmark: Kill Identity

```
kill_identity {
    let r in 1 .. T::MaxRegistrars::get();
    let s in 1 .. T::MaxSubAccounts::get();
    let x in 1 .. T::MaxAdditionalFields::get();
    let target: T::AccountId = account("target", SEED, SEED);
    let target_origin: <T as frame_system::Trait>::Origin = RawOrigin::Signed(target.clone()).into();
    let target_lookup: <T::Lookup as StaticLookup>::Source = T::Lookup::unlookup(target.clone());
    let _ = T::Currency::make_free_balance_be(&target, BalanceOf::<T>::max_value());
    Identity::<T>::provide_judgement( /* snip */ )?;
    }

    let call = Call::<T>::kill_identity(target_lookup);
    let force_origin = T::ForceOrigin::successful_origin();
    ensure!(IdentityOf::<T>::contains_key(&target), "Identity not set");
}: { call.dispatch_bypass_filter(force_origin)? }
verify {
    ensure!(!IdentityOf::<T>::contains_key(&target), "Identity not removed");
}
```

- Set up an account with the appropriate funds.
- Note this is just like writing runtime tests!

Benchmark: Kill Identity

```
kill_identity {
    let r in 1 .. T::MaxRegistrars::get();
    let s in 1 .. T::MaxSubAccounts::get();
    let x in 1 .. T::MaxAdditionalFields::get();

    let target: T::AccountId = account("target", SEED, SEED);
    let target_origin: <T as frame_system::Trait>::Origin = RawOrigin::Signed(target.clone()).into();
    let target_lookup: <T::Lookup as StaticLookup>::Source = T::Lookup::unlookup(target.clone());
    let _ = T::Currency::make_free_balance_be(&target, BalanceOf::<T>::max_value());

    let info = create_identity_info::<T>(x);
    Identity::<T>::set_identity(target_origin.clone(), info)?;
    let _ = add_sub_accounts::<T>(&target, s)?;

}: { call.dispatch_bypass_filter(force_origin)? }
verify {
    ensure!(!IdentityOf::<T>::contains_key(&target), "Identity not removed");
}
```

- Using some custom functions defined in the benchmarking file:
  - Give that account an Identity with x additional fields.
  - Give that Identity s sub-accounts.

Benchmark: Kill Identity

```
kill_identity {
    let r in 1 .. T::MaxRegistrars::get();
    let s in 1 .. T::MaxSubAccounts::get();
    let x in 1 .. T::MaxAdditionalFields::get();

    let target: T::AccountId = account("target", SEED, SEED);
```

- Add r registrars.
- Have all of them give a judgement to this identity.

```
    // User requests judgement from all the registrars, and they approve
    add_registrars::<T>(r)?
    for i in 0..r {
        Identity::<T>::request_judgement(target_origin.clone(), i, 10.into())?;
        Identity::<T>::provide_judgement( /* snip */ )?;
    }

    let call = Call::<T>::kill_identity(target_lookup);
    let force_origin = T::ForceOrigin::successful_origin();
    ensure!(IdentityOf::<T>::contains_key(&target), "Identity not set");
}: { call.dispatch_bypass_filter(force_origin)? }
verify {
    ensure!(!IdentityOf::<T>::contains_key(&target), "Identity not removed");
}
```

Benchmark: Kill Identity

```
kill_identity {
    let r in 1 .. T::MaxRegistrars::get();
    let s in 1 .. T::MaxSubAccounts::get();
    let x in 1 .. T::MaxAdditionalFields::get();
```

- First ensure statement verifies the “before” state is as we expect.
- We need to use our custom origin, so we “dispatch” the function call.
  - For normal cases, the syntax is even more simple, but this shows that you can place **any** logic in a benchmark, and we can measure it… not just extrinsics.
- Verify block ensures our “final” state is as we expect.

```
    let call = Call::<T>::kill_identity(target_lookup);
    let force_origin = T::ForceOrigin::successful_origin();
    ensure!(IdentityOf::<T>::contains_key(&target), "Identity not set");
}: { call.dispatch_bypass_filter(force_origin)? }
verify {
    ensure!(!IdentityOf::<T>::contains_key(&target), "Identity not removed");
}
```

Executing The Benchmark

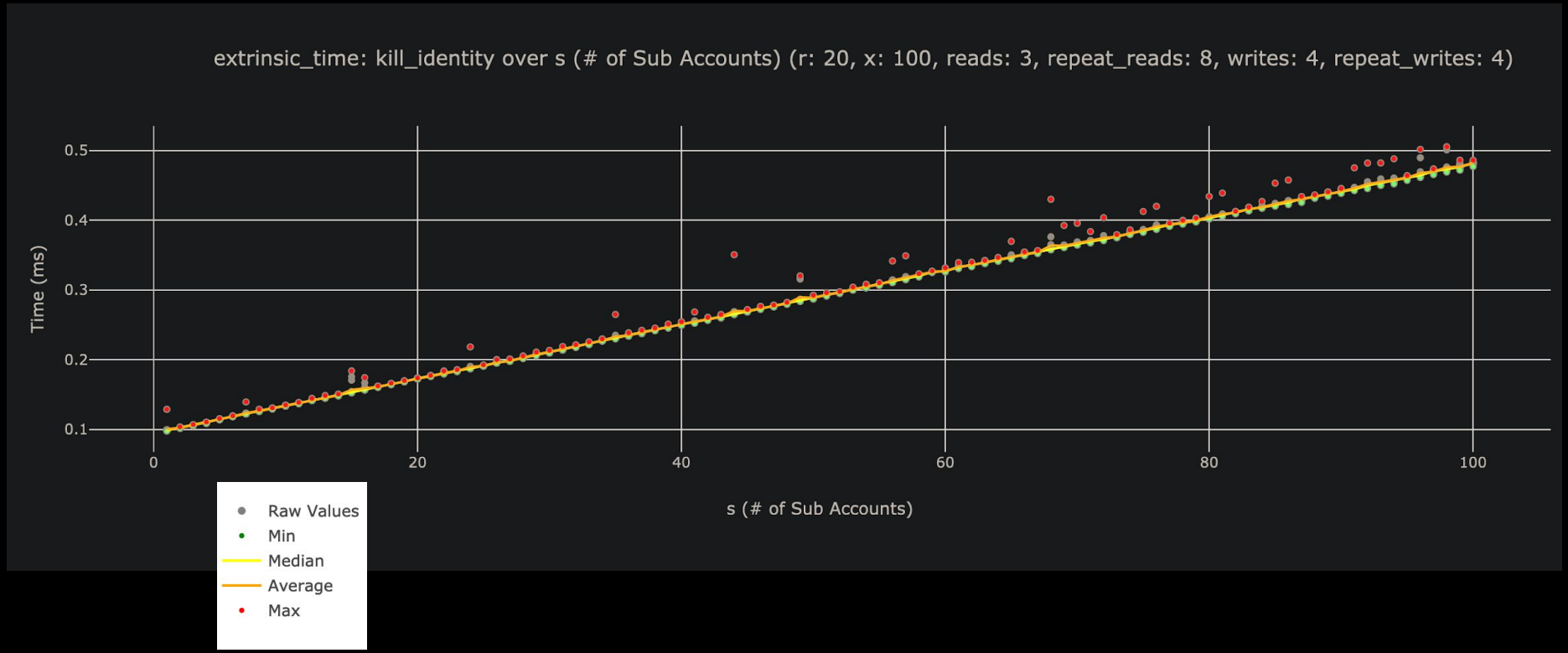
```
./target/release/substrate benchmark \
    --chain dev \                # Configurable Chain Spec
    --execution=wasm \           # Always test with Wasm
    --wasm-execution=compiled \  # Always used `wasm-time`
    --pallet pallet_identity \   # Select the pallet
    --extrinsic kill_identity \  # Select the extrinsic
    --steps 50 \                 # Number of steps across component ranges
    --repeat 20 \                # Number of times we repeat a benchmark
    --raw \                      # Optionally output raw benchmark data to stdout
    --output ./                  # Output results into a Rust file
```

```
./target/release/substrate benchmark \
    --chain dev \
    --execution=native \         # Use native to access debug logs
    --pallet pallet_identity \
    --extrinsic kill_identity \
    --steps 50 \                 # Still need steps, but repeat can be 1.
    --log benchmark              # Output DB operation logs to stderr
```

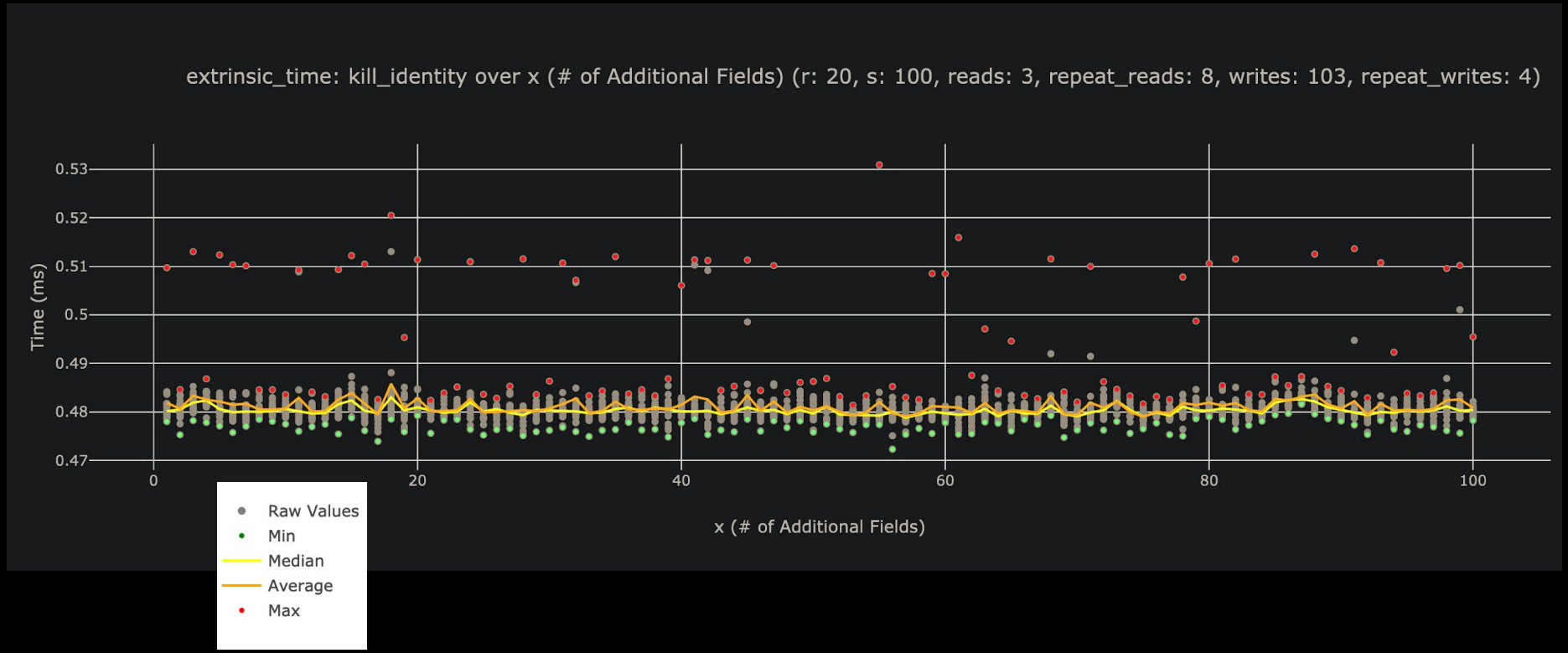
OPTIONAL: Only if you want to peek closer at the DB logs.

Results: Extrinsic Time vs. # of Registrars

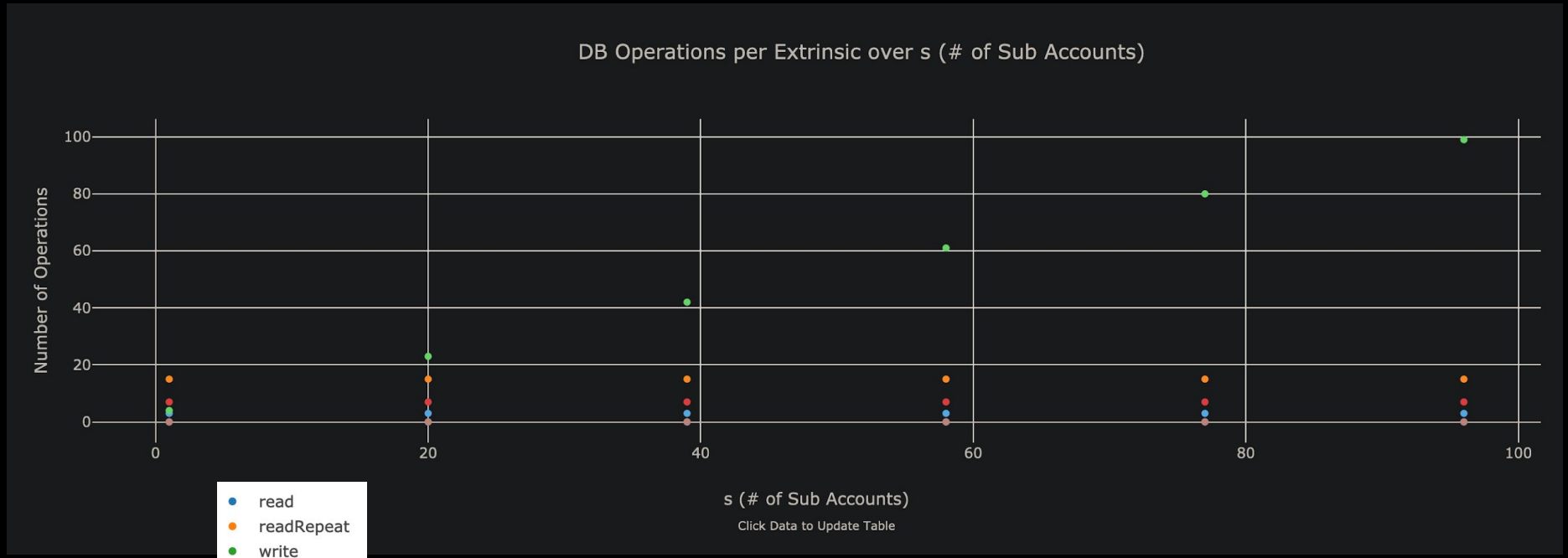
Results: Extrinsic Time vs. # of Sub-Accounts



Results: Extrinsic Time vs. # of Identity Fields



Result: DB Operations vs. Sub Accounts



NOTE: Other components had constant DB Operations

Autogenerated Weight Formula

```
fn kill_identity(r: u32, s: u32, x: u32, ) -> Weight {
   (123_199_000 as Weight)
       .saturating_add((71_000 as Weight).saturating_mul(r as Weight))
       .saturating_add((5_730_000 as Weight).saturating_mul(s as Weight))
       .saturating_add((2_000 as Weight).saturating_mul(x as Weight))
       .saturating_add(T::DbWeight::get().reads(3 as Weight))
       .saturating_add(T::DbWeight::get().writes(3 as Weight))
       .saturating_add(T::DbWeight::get().writes(
             (1 as Weight).saturating_mul(s as Weight))
       )
}
```

WeightInfo Integration

```
#[weight = T::WeightInfo::kill_identity(
   T::MaxRegistrars::get().into(),  // R
   T::MaxSubAccounts::get().into(),  // S
   T::MaxAdditionalFields::get().into(),  // X
)]
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) -> DispatchResultWithPostInfo
{
   T::ForceOrigin::ensure_origin(origin)?;

   /* snip */

   Self::deposit_event(RawEvent::IdentityKilled(target, deposit));

   Ok(Some(T::WeightInfo::kill_identity(
       id.judgements.len() as u32, // R
       sub_ids.len() as u32, // S
       id.info.additional.len() as u32 // X
   )).into())
}
```

WeightInfo Integration

```
#[weight = T::WeightInfo::kill_identity(
   T::MaxRegistrars::get().into(),  // R
   T::MaxSubAccounts::get().into(),  // S
   T::MaxAdditionalFields::get().into(),  // X
)]
```

- Use the WeightInfo function as the weight definition for your function.
- Note that we assume absolute worst case scenario to begin since we cannot know these specific values until we query storage.

```
sultWithPostInfo

   Ok(Some(T::WeightInfo:: kill_identity(
      id.judgements. len() as u32, // R
      sub_ids. len() as u32, // S
      id.info.additional. len() as u32 // X
   )).into())
}
```

WeightInfo Integration

```
#[weight = T::WeightInfo::kill_identity(
   T::MaxRegistrars::get().into(),  // R
   T::MaxSubAccounts::get().into(),  // S
   T::MaxAdditionalFields::get().into(),  // X
)]
fn kill_identity(origin, target: <T::Lookup as StaticLookup>::Source) -> DispatchResultWithPostInfo
{
```

- Then we return the *actual* weight used at the end!
- We use the same WeightInfo formula, but using the values that we queried from storage as part of executing the extrinsic.

```
  Ok(Some(T::WeightInfo:: kill_identity(
      id.judgements. len() as u32, // R
      sub_ids. len() as u32, // S
      id.info.additional. len() as u32 // X
  )).into())
}
```

Things we tried
Things we learned

Isolating DB Benchmarks (PR #5586)

We tried...

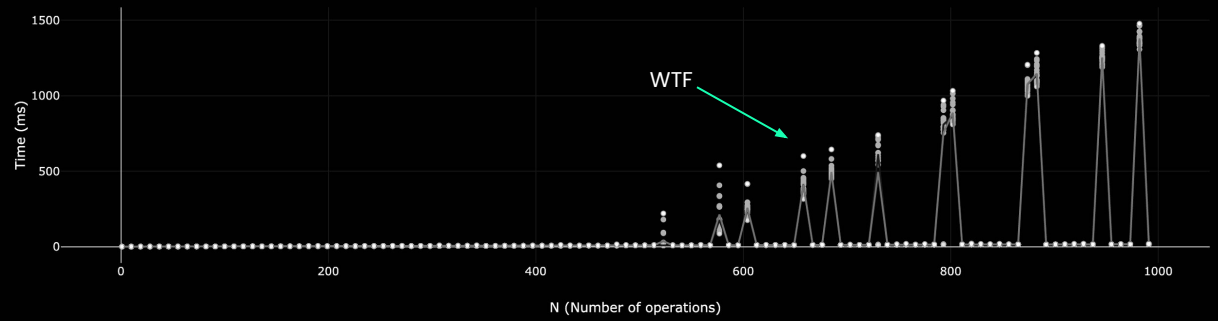
To benchmark the entire extrinsic, including the weight of DB operations directly in the benchmark. We wanted to:

- Populate the DB to be “full”
- Flush the DB cache
- Run the benchmark

We learned...

RocksDB was too inconsistent to give reproducible results, and really slow to populate.

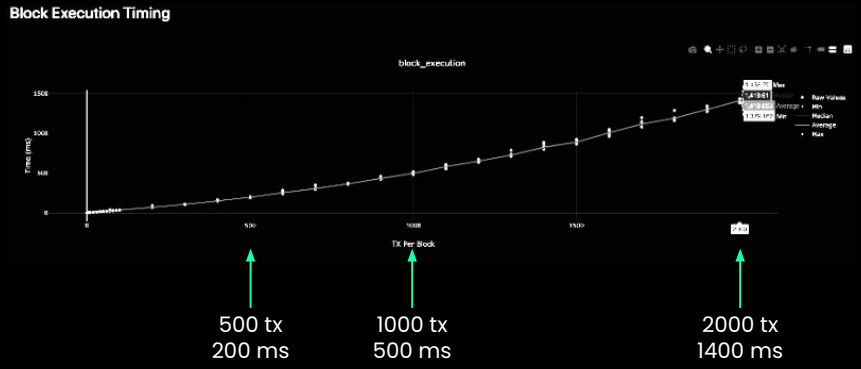
So we use an in-memory DB for benchmarking.



Fixing Nonlinear Events (PR #5795)

We tried...

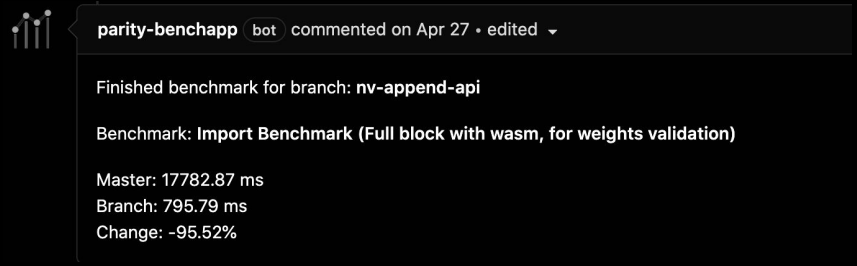
Executing a whole block, increasing the number of txs in each block. We expected to get linear growth of execution time, but in fact it was superlinear!



We learned...

Each time we appended a new event, we were passing the growing event object over the Wasm barrier.

We updated the append api so only new data is pushed.



Enabling Weight Refunds (PR #5584)

We tried...

To assign weights to all extrinsics for the absolute worst case scenario in order to be safe.

In many cases, we cannot know accurately what the weight of the extrinsic will be without reading storage... and this is not allowed!

We learned...

That many extrinsics have a worst case weight much different than their *average* weight.

So we allow extrinsics to return the actual weight consumed and refund that weight and any weight fees.

Customizable Weight Info (PR #6575)

We tried...

To record weight information and benchmarking results directly in the pallet.

```
#[weight = 45_000_000 +
T::DbWeight::get().reads_writes(1,1)]
```

We learned...

This was hard to update, not customizable, and not accurate for custom pallet configurations.

So we moved the weight definition into customizable associated types configured in the runtime trait.

```
#[weight = T::WeightInfo::transfer()]
```

Best Practices & Common Patterns

Initial Weight Calculation Must Be Lightweight

- In the TX queue, we need to know the weight to see if it would fit in the block.
- This weight calculation must be lightweight!
- No storage reads!

Example:

- Transfer Base: ~50 µs
- Storage Read: ~25 µs

Set Bounds and Assume the Worst!

- Add a configuration trait that sets an upper bound to some item, and in weights, initially assume this worst case scenario.
- During the extrinsic, find the actual length/size of the item, and refund the weight to be the actual amount used.

Separate Benchmarks Per Logical Path

- It may not be clear which logical path in a function is the “worst case scenario”.
- Create a benchmark for each logical path your function could take.
- Ensure each benchmark is testing the worst case scenario of that path.

Comparison Operators in the Weight Definition

```
#[weight =
    T::WeightInfo::path_a()
    .max(T::WeightInfo::path_b())
    .max(T::WeightInfo::path_c())
]
```

Keep Extrinsics Simple

- The more complex your extrinsic logic, the harder it will be to accurately weigh.
- This leads to larger up-front weights, potentially higher tx fees, and less efficient block packing.

Use Multiple Simple Extrinsics

- Take advantage of UI/UX, batch calls, and similar downstream tools to simplify extrinsic logic.

Example:

- Vote and Close Vote (“last vote”) are separate extrinsics.

Minimize Usage of On Finalize

- on_finalize is the last thing to happen in a block, and must execute for the block to be successful.
- Variable weight needs at can lead to overweight blocks.

Transition Logic and Weights to On Initialize

- on_initialize happens at the beginning of the block, before extrinsics.
- The number of extrinsics can be adjusted to support what is available.
- Weight for on_finalize should be wrapped into on_initialize weight or extrinsic weight.

Understand Limitations of Pallet Hooks

- A powerful feature of Substrate is to allow the runtime configuration to implement pallet configuration traits.
- However, it is easy for this feature to be abused and make benchmarking inaccurate.

Keep Hooks Constant Time

- Example: Balances hook for account creation and account killed.
- Benchmarking has no idea how to properly set up your state to test for any arbitrary hook.
- So you must keep hooks constant time, unless specified by the pallet otherwise.

What's next?

How you might be able to contribute!

- Better regression analysis
- More insight into DB Ops
  - Optimization hints based on duplicate reads/writes
- Improved Automation
- Integration into FRAME macros

Questions?

shawntabrizi@parity.io | @shawntabrizi