Performance Engineering in Web App

IF3110 Web-Based Application Development
School of Electrical Engineering and Informatics
Institut Teknologi Bandung

Performance Engineering

- Process by which software is tested and tuned
  with the intent of realizing the required performance
- Performance categories
  - Speed – Scalability – Stability
- Objective of Performance engineering:
  - Increase revenue by ensuring optimum system
  - Optimize the provisioning & utilization of infrastructure
  - Improve availability by resolving performance & scalability
  - Reduce maintenance costs
  - Avoid system failure requiring scrapping & writing off development effort
  - Self impressive facts about the release

Example-1

- The system was run out the bandwidth because many tax-payers reported their statement close by the deadline (31st March 2018)
- The administrators suggested these issues due to the network capacity.

Example-2

- Many users cannot register or waited too long because the system had been scheduled to process (bogus) applications

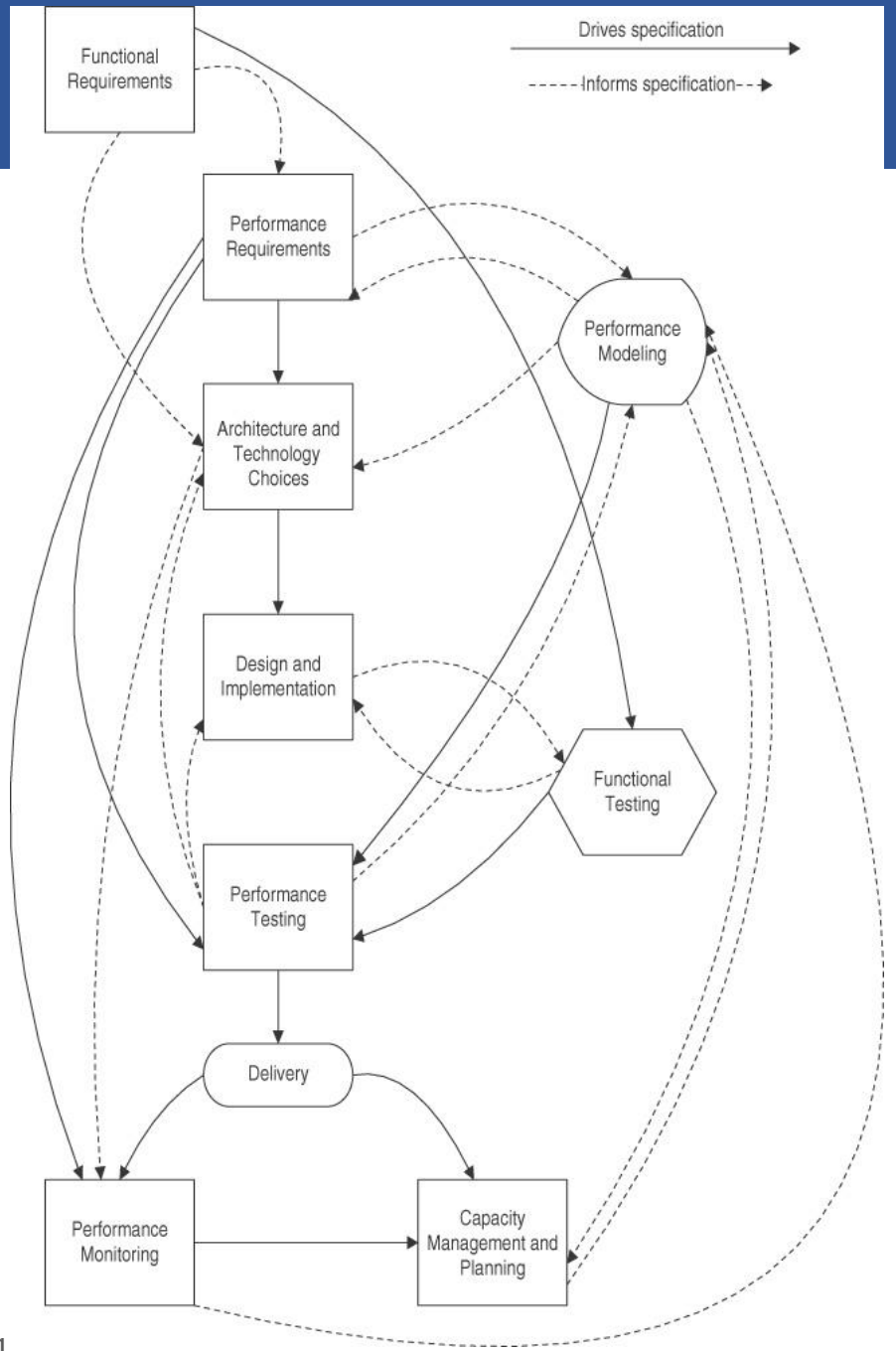
Example-3

- An online securities trading system must be able to handle large numbers of transactions per second, especially in a volatile market with high trading volume. A brokerage house whose system cannot do this will lose business very quickly, because slow execution could lead to missing a valuable trading opportunity.
- An online banking system must display balances and statements rapidly. It must acknowledge transfers and the transmission of payments quickly for users to be confident that these transactions have taken place as desired.

Part of Performance Engineering

- Performance testing & sizing
  - Regression Test, System Load testing, System Benchmarking
- Performance tuning and optimization
  - Optimal tuning guidelines for production setup – JVM, Pool, logs, App/Web server, DB, OS, etc.
  - Effectively applying skills, technologies and tools
- System diagnostics
  - End to end profiling, recommendations for overall system scalability
- Capacity planning
  - Determine expected production capacity, Facilitate capacity management

Performance Engineering and SW Engineering



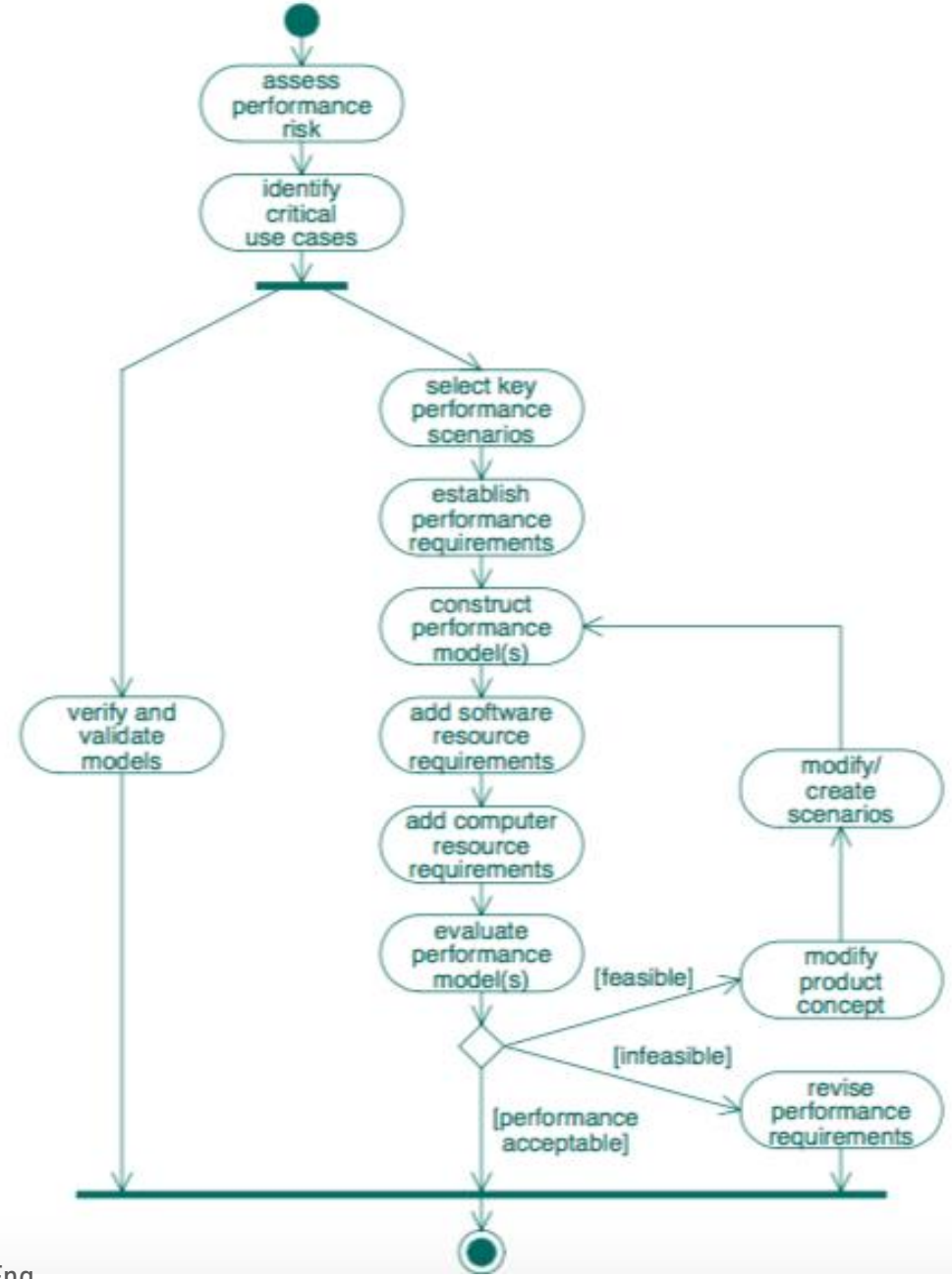
Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice
By André B. Bondi

Approach to Performance Eng.

- Begins early, frequency matches system criticality
- Often find architecture & design alternatives with lower resource requirements
- Select cost-effective performance solutions early

Modeling Performance

- Assess performance risk
- Identify critical use cases
- Select key performance scenarios
- Establish performance requirements
- Construct performance models
- Determine software resource requirements
- Add computer resource requirements
- Evaluate the models
- Verify and validate the models



Performance Scenarios/Use Case

- Workload identification and characterization
- Benchmark scenarios
- Precise performance requirements
  - Response time
  - Throughput
  - Resource utilizations/budgets

Modeling Strategies

- Simple-Model Strategy
  - start with the simplest possible model that identifies problems with the system architecture, design or implementation plans
- Best and Worst-case Strategy
  - use best- and worst-case estimates of resource requirements to establish bounds on expected performance and manage uncertainty in estimates
- Adapt-to-Precision Strategy
  - match the details represented in the models to your knowledge of the software processing details

Verify & Validate the Model

- Verify the model data vs reality
- Validate the results in reality with the ones from the model
- Begin early in the life cycle and continue throughout life cycle
- Check for model omissions!

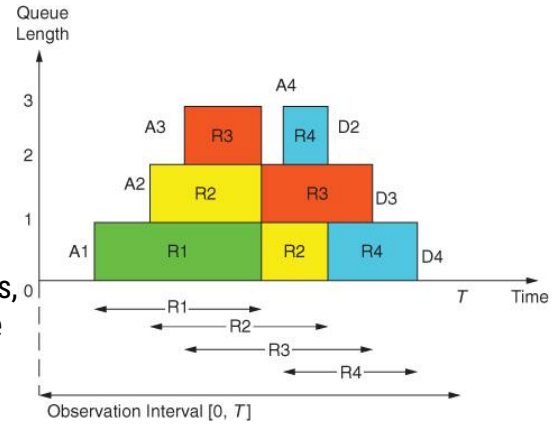
Basic Performance Laws

- **The Utilization Law** states that the average utilization (U) of a single server is its throughput (T) multiplied by the average service time (S)

$$U = T.S < 1$$

- **Little's Law** relates the average response time (R), average throughput (T), and average queue length (q). Note that these are all time-averaged quantities.

$$q = T.R$$



Foundations of Software and System Performance Engineering: Process, Performance Modeling, Requirements, Testing, Scalability, and Practice
By André B. Bondi

Performance Metrics

- Describe the performance of a system in terms that are commonly understood and unambiguously defined.
  - For: engineering and business perspectives.
- Common fallacies: ambiguities in quantitative descriptions, or because system performance is described in terms of quantities that cannot be measured in the system of interest.
- Performance is described in terms of quantities known as *metrics*.
  - A metric is defined as a standard of measurement

Performance Metrics - Examples

- response time is a standard measure of how long it takes a system to perform a particular activity. This metric is defined as the difference between the time at which an activity is completed & the time at which it was initiated. e.g., absolute/world time vs CPU time
- Response Time = Wait Time + Service time
- The average device utilization is a standard measure of the proportion of time that a device, such as a CPU, memory, disk, or other I/O device, is busy.
  - the length of time of measurement should be stated; too long intervals → hide fluctuations XOR too short intervals → reveal them.
- The average waiting time of a server/device
- The average throughput of a system is the rate at which the system completes units of a particular activity.

Each application domain might have its **own** performance metrics

Properties of a Good Performance Metric

- Linearity
- Reliability
- Repeatability
- Ease of Measurement
- Consistency
- Independence

Challenges in PE

- Right tool selection
- Identifying the problems: debugging, bottleneck, intermittent problems
- Appropriate Hardware setup
- Governing the behaviour of simulated users according to a set of policies
- Consistent, repeatable actions
- Accurate measurement of response times
- Generation and comprehensive analysis of results

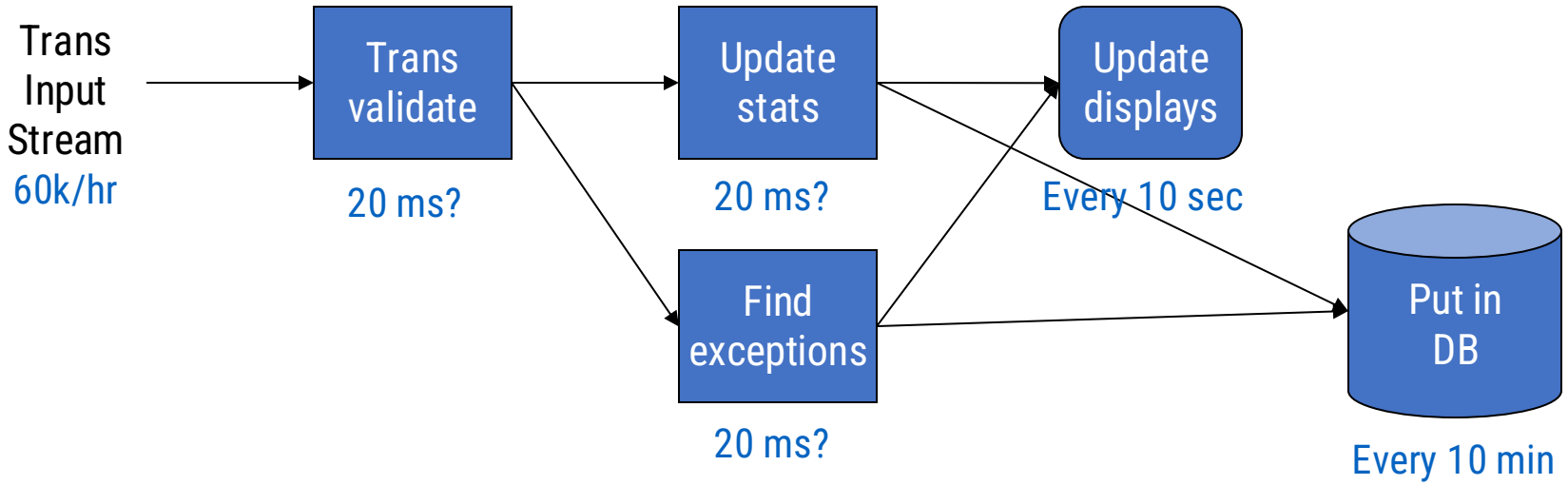
Illustration on Performance Analysis

Here was the original example

- You have a system that monitors economic transactions for ecommmerce.co.id
- Let's look at critical use cases / scenarios:
  - It sees 60,000 transactions per hour (peak hour).
  - Each validated transaction updates status and activity information in the memory of a server.
  - You have five displays for people watching. They see exception transactions and statistics.
  - Oh, and exception transactions need to be shown, too.
  - These screens should automatically update every 10 seconds.
  - Every 10 minutes the in-memory info is saved to disk, using SQL-Server.

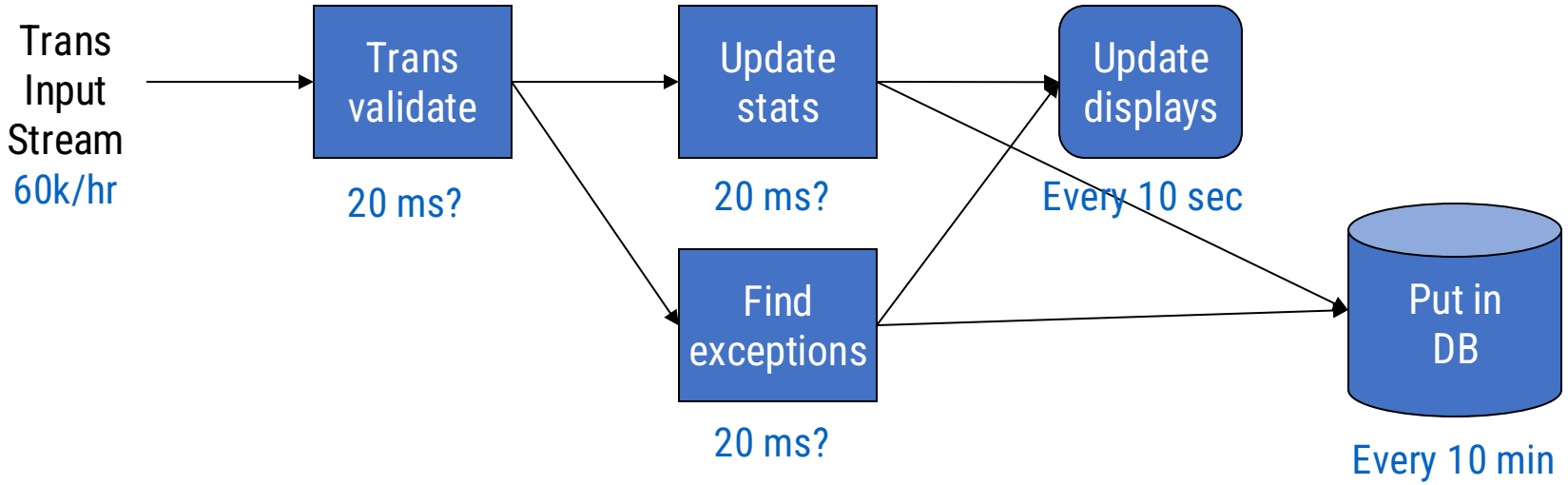
our first analysis

- Design looks something like this figure.
- 60,000 trans/hr = 1000 trans/min = 16.7 trans/sec = 60 ms/trans.
- **Naïvely assume** each of the 3 functions (validation, stats update, find exceptions) on a trans takes equal time.
- So, they each have to be done in 20 ms.
- But there are also two performance "lumps" –
  - Updating the 5 displays every 10 sec, and
  - Writing the memory data to disk every 10 min!



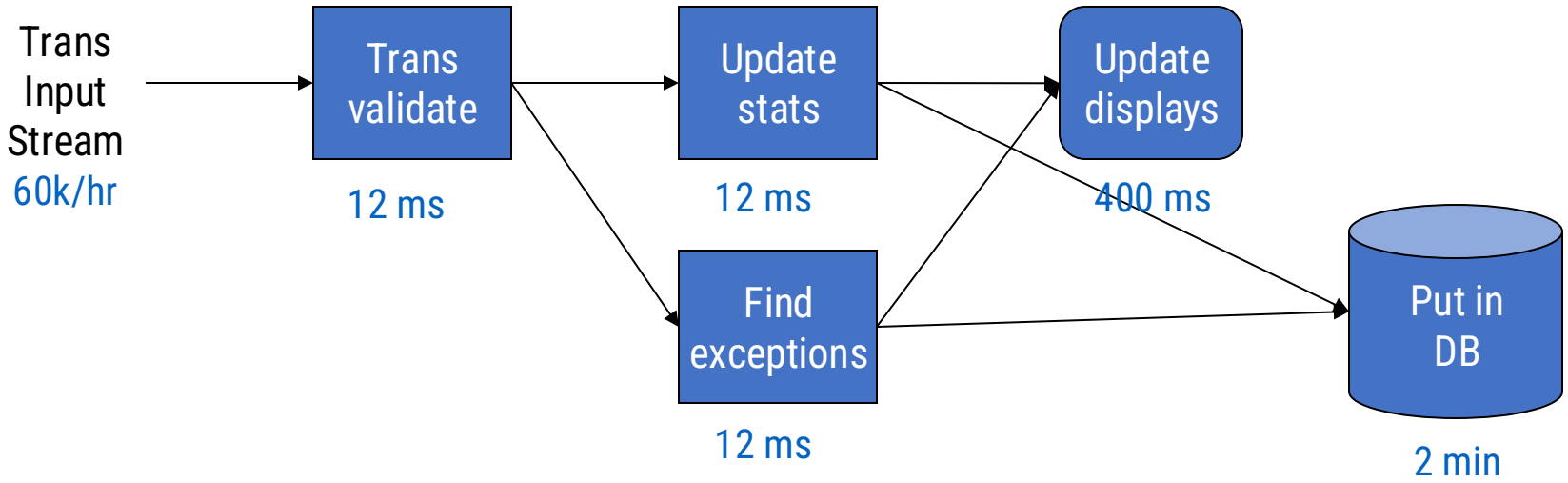
So… is it our proposals?

- Can you assume that these “lumps” can be tuned out of the system during testing?
  - Why or why not?



When we did – this was a better answer:

- Not unless you're used to dealing with such things already!
- If not, better budget for them, too:
  - Divide the 60 ms/trans by 5, not by 3.
  - Each of the 3 original functions shown gets 12 ms/trans.
  - Display refresh gets 1/5 of every CPU second, or 200 ms. So 5 displays refreshed every 10 sec = 1 every 2 sec. Each display refresh gets a 400 ms budget.
  - DB write gets 1/5 of every CPU second, or 200 ms, also. Over 10 min, it then gets 200 * 60 * 10 ms = 2 min of CPU time. But, this had better be distributed evenly!



But…we then added this info:

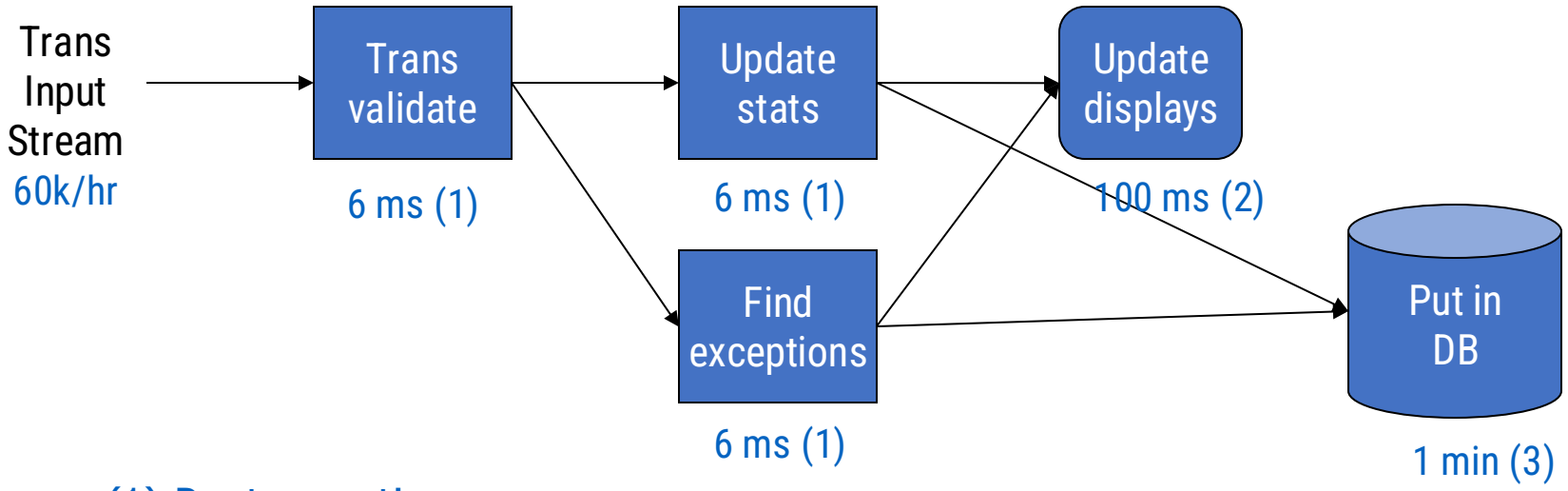
- This is still optimistic!
- It assumes you have all the CPU time for your application, and
- It assumes your transactions aren't "lumpy," and
- That the system won't grow.
- A more conservative start would be to cut all the targets in half, on the previous page.

So, how to proceed?

1. Being “conservative,” and reallocating what we had, to use only 50% of the whole CPU time.
2. How to handle the fact that “Trans validate” code is now estimated as only half as complex as the code in the “Update stats” and “Find exceptions” routines.
3. Saving all the statistics to the DB must be one “synchronized” action every 10 min. This is now estimated to take 5 sec. The remaining DB tasks, however, can be distributed evenly over each 10 min interval.

Let’s do the “conservative” one:

1. Being “conservative,” and reallocating what we had, to use only 50% of the whole CPU time, we get half of each figure from before:



(1) Per transaction.
(2) Per second, for all 5 displays.
(3) Every 10 min, for the DB.

Reallocating the transaction budget times:

2. How to handle the fact that “Trans validate” code is now estimated as only half as complex as the code in the “Update stats” and “Find exceptions” routines:

You could have made different assumptions here. Let’s assume you *only* needed to reallocate the budgets for the 3 routines handling the transactions. We had:

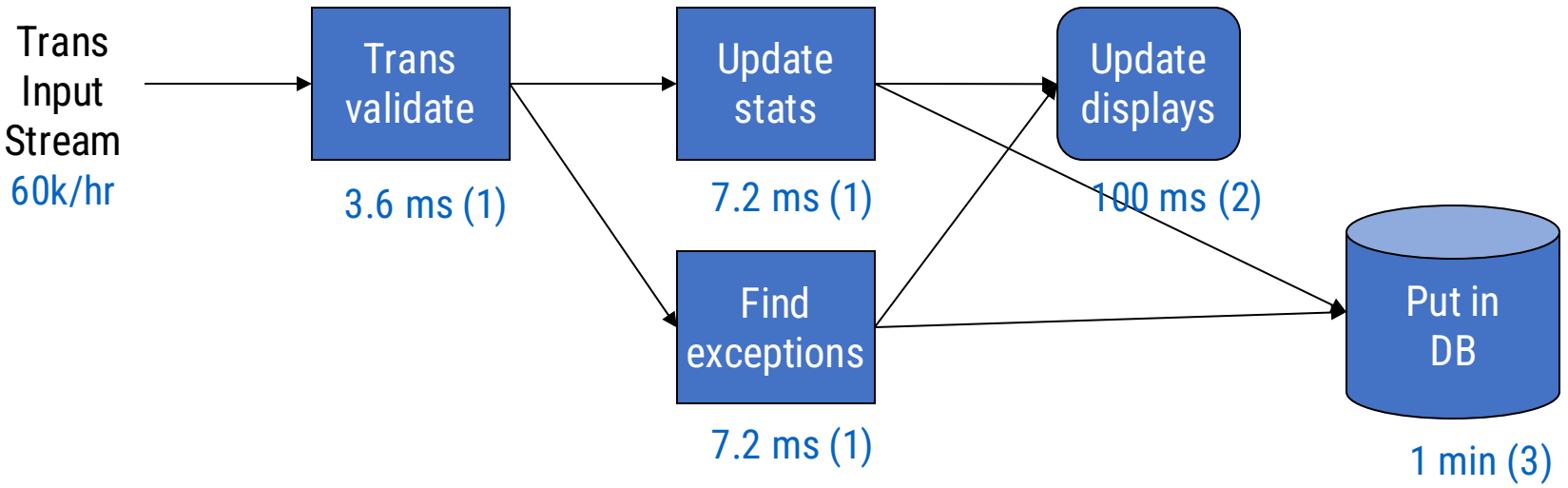
6 ms + 6 ms + 6 ms = 18 ms total. But we now want:

x ms + 2*x ms + 2*x ms = 18 ms, where x = Trans validate’s budget.

So, 5 * x = 18, and x = 3.6 ms. The other two transaction-related routines each get a new budget of 7.2 ms (instead of 6 ms).

And the whole thing now looks like this:

- (Again, we assumed only the transaction-handling budgets were affected here.)



(1) Per transaction.
(2) Per second, for all 5 displays.
(3) Every 10 min, for the DB.

Now, what about the DB part?

3. Saving all the statistics to the DB must be one “synchronized” action every 10 min. This is now estimated to take 5 sec. The remaining DB tasks, however, can be distributed evenly over each 10 min interval.

This asks us to handle a “lumpy distribution problem.” Let’s assume that the DB statistics action is “synchronized” by locking the table it is writing to the DB.

Let’s also figure we still get 1 CPU minute every 10 minutes for *all* the DB saving activities, so leaves a 60 – 5 = 55 sec budget for the non-stat DB actions.

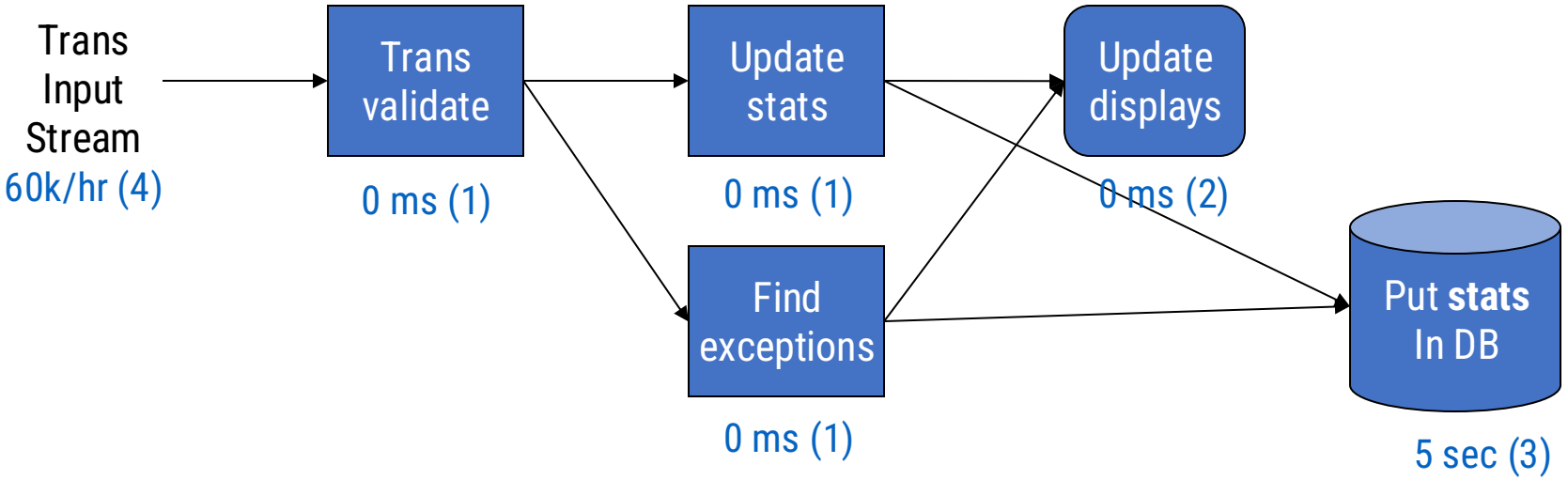
If we get half the CPU time overall for our application, the DB stat action thus “locks out” the other parts of our application for 2*5 = 10 clock seconds in a row, and this happens every 10 minutes. (Or, at least the parts of our app which update the DB from each transaction.)

So, what you need to do is…

- Show two pictures of the budgets –
  - How they look when this synchronized DB **stats** update is happening, and
  - How they look when it isn't happening.
- That handles the lumpiness.

Here's one solution…

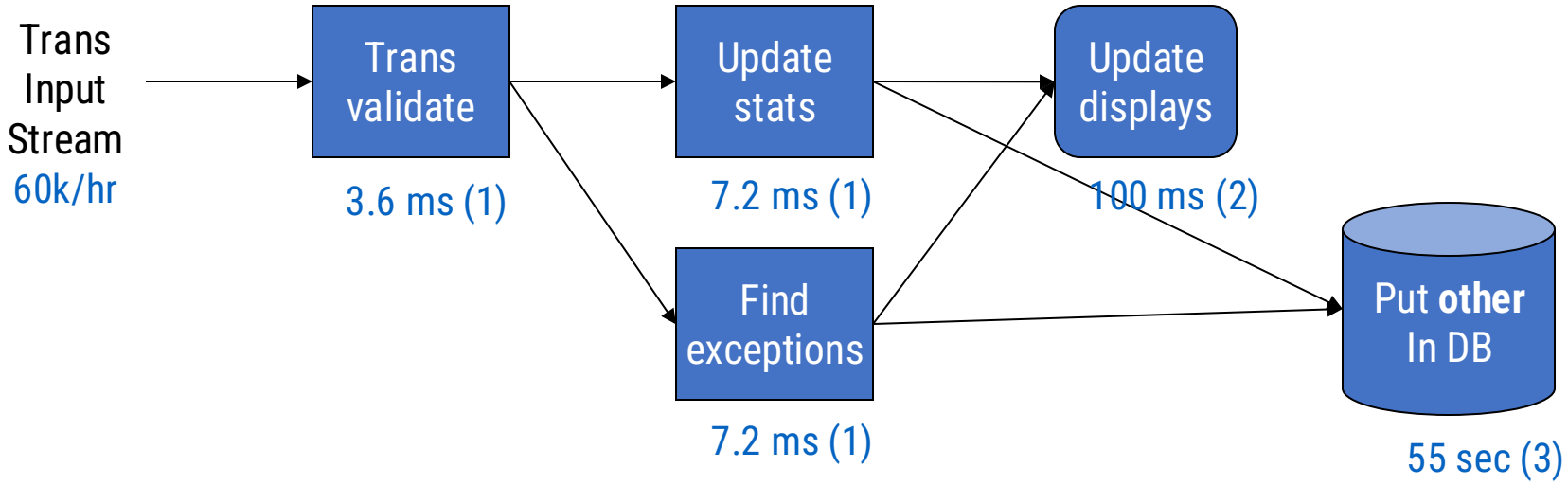
a. During the 10 sec of time the DB **stats** update goes on:



(1) Per transaction.
(2) Per second, for all 5 displays.
(3) Every 10 min, for the DB.
(4) Assume that transactions simply back up in buffers!

And…

b. During the rest of the time:



(1) Per transaction.
(2) Per second, for all 5 displays.
(3) Every 10 min, for the DB.

In the case of Web App

Noteworthy

- Look at CPU time.
- Other things often resource budgeted and tracked:
  - Memory space
  - Disk I/O time
  - Communications time
- Things related to Page Load
  - Resource number and size
  - Caching strategy
  - Render time
  - Network-related (DNS lookup, Network/TLS things, Latency, Bandwidth)

Noteworthy

- Things to worry with Database
  - Put business intelligence/analytics away from the transaction processing
  - Beware with the cost of relationships (JOIN, 3ND)
  - Use the database lock wisely
  - Use transaction if it is really needed
  - Avoid SELECT before UPDATE
  - Think twice before
    - SELECT * from XXX
    - SORT BY
    - GROUP BY
  - Put away old data – purge vs archive

Noteworthy

- Things related to Web App Design
  - For scalability – Async, Loosely coupled, Stateless
  - For testability – Interface-first, Test-driven Dev, Automated Testing
  - For observability - Log, Metrics, Traces